TIE-20306 Principles of Programming Languages, autumn 2019

Last change 27.11.2019 17:19.


```
This page change history:
2019-11-22: First public version
2019-11-27: Added description of provided helper code. Changed requirements of interpretation levels 1 and 3 a
```

Project work, phase 4

Semantic checks and/or running the program

Using the syntax tree it is now time for semantic checks and running the program. For full credits you DO NOT have to implement everything listed here. One can easily think about dozens of different checks and implementing them all would be too much work for this project.

So select the items you are interested in and concentrate creating a clearly documented, easy to read implementation of them. You might concentrate on hunting semantic errors or running the program.

A fully functional example on how to do semantic checks and a simple interpreter using our syntax tree can be found here.

- File `semantics_common.py` has utility functions for the symbol table and visiting nodes of the syntax tree (which you can use in your own phase 4, too). The provided function and class are described at the end of this page.
- File `semantics_check.py` has functions to perform semantic checks by visiting all tree nodes, recognizing node types and checking them. Running this file performs the semantic checks for a given program.
- File `semantics_run.py` contains a simple interpreter which recursively evaluates the syntax tree. Running this file performs semantic checks, prints out the syntax tree and then runs the interpreter.

Semantics of the language

This section describes the intended semantics (= meaning) of the language constructs. Please note that this is just a description, you don't have to implement all of these! The following sections describe what semantic checks and functionality to implement.

These are just quidelines. You decide the exact semantics yourself (and document them in your submission).

Datatypes

- In binary operations (expressions), both operands can be integers and result is an integer.
- Integer and string cannot be used in the same expression.
- Strings can be added together (catenated)
- Two tuples can be catenated together using the ++ operator
- `[A..B]` creates a tuple with integer values from A to B
- `[N**M]` creates a tuple with N instances of value M
- `[a, b, c]` creates a tuple with values a, b, and c (this is probably obvious)

Variables

- Choice 1: Each variable can only be defined once (i.e., there's no assignment to an existing variable, just initialization of a new variable, we have a functional programming language) *This is a possible semantic check*.
- Choice 2: Alternatively you can choose that constants can only be defined once, others can be be assigned to multiple times. Please remember to document your choice! *This is a possible semantic check*.
- Variables can only be used after they have been defined (just as in most programming languages) *This is a possible semantic check*.
- You don't *have to* implement scoping. I.e., it's ok to assume that all variables are "global" regardless where they are defined. This implies that all variables and function parameters have different names so that there's no overlap.
- A little more challenging alternative (which gives you bonus points) is to have all variables still as global, but check that local variables and parameter names can only be used in the function body.

- Final most challenging alternative is to implement scoping, i.e. global variables and local varibles (or parameters) can have the same name, but both still refer to a different variable.

Functions

- A function parameter name can be used only inside the respective function. *This is a possible semantic check*.
- The language does not support recursion, i.e. it's not possible for two calls to the same function to be active at the same time. (This makes it unnecessary to implement dynamic activation records, etc. You are allowed to support recursion, if you want, though, which gives you bonus points.) (implementing recursion requires some sort of activation records for local variables, which may be quite complicated to do, so we do not recommend it (too much work for this course)).
- Each parameter name is unique in the program, see the "Variables" section for alternatives
- Returning a value with '=' returns a single integer/string value, which means that function can only be called in places where a single value is allowed *This is a possible semantic check*.
- Returning a value with '!=' returns a tuple, which means that function can only be called in places where a tuple is allowed *This is a possible semantic check*.

Pipes

- Piping a tuple to '+' produces a tuple containing the sum of the tuple's elements. For example, "`[2,3,4] | +`" results in `[9]`.
- Piping a tuple to '*' produces a tuple containing the product of the tuple's elements. For example, "`[2,3,4] | *`" results in `[24]`.
- Piping a tuple to a function calls that function with the tuple's elements as function's arguments (the size of the tuple has to be the same as the parameter count). If the function returns a tuple, that tuple is the result of piping. If the function returns a single value, a tuple containing the value is the result. For example, "`[1,2,3] | Func`" means "`Func[1,2,3]`". *This is a possible semantic check*.
- Piping a tuple to "each:" takes as many elements from the tuple as the function takes parameters, calls the function with those, then repeats this for the next batch of elements, and so on. The number of elements in the tuple must be a multiple of how many parameters the function takes (i.e., there must be no "leftover" tuple elements). The result is the results of function return values catenated together. For example, if `Func` takes 2 parameters, "`[1,2,3,4] | each:Func`" is equal to "`Func[1,2] ++ Func[3,4]`". *This is a possible semantic check*.

Built-ins

- Function 'Print' takes any number of arguments and prints out their string representation. (HINT: arbitrary number of parameters is not a hard problem for the compiler/interpreter - arguments are already in 'function_call' list in the syntax tree). The return value of Print is a tuple containing the arguments
- Code "`select:n[<tuple>]`" returns the nth element of the tuple (i.e., "`select:1...`" returns the first element). The index of the element must be in range, i.e. 1 <= n <= size-of-tuple. *This is a possible semantic check*.
- Invent your own built-in funtions, if you wish!

Semantic checks

In this assigment, semantic checks are performed **on the syntax tree** (i.e., before running the program, even if you implement the interpreter part). In many programming languages ALL the checks are run and results printed out. However, it's easier to stop at the first semantic error (since recovering from the error and continuing is often quite complicated), so that's what we'll do in this assignment.

In the Semantics of the language section above, ideas for possible semantic checks are marked with "*This is a possible semantic check*.". However, there are more possible semantic checks that could be performed, so it's ok to also include your own, if you wish. The list below lists the mentioned semantic checks in the estimated order of easiness (the easiness of course depends somewhat on your design choices):

1. Variables, functions, and parameters have to be defined before being used, no double definitions allowed.
2. Parameters are only allowed to appear in the function in which they have been defined.
3. The number of actual parameters in a function call has to match the number for formal parameters in the function definition.
4. If a function returns a single value (with =), it can only be used in a place where a single value is ok. If a function returns a tuple (with !=), it can only be used in a place where a tuple is ok.
5. When piping a tuple to a function or "each:", the size of the tuple has to agree with the number of parameters. Similarly, in "select:" the element number has to be within the tuple size. *This semantic check is a little more challenging that most others, since it requires you to track the sizes of tuples in the syntax tree*.

Implementing semantic checks

Probably easiest way to implement semantic checking of the syntax tree is to use the provided visitor pattern based function `visit_tree()`, as well as provided simple data structure `SemData`. The visitor can be used to perform checking passes over the tree, selecting which types of node to act on. The SemData class can be used to store the symbol table as well as other information needed for semantic checking, as well as passing that information along when performing the checks. For more hints on how to implement phase 4, please watch the 26.11. lecture recording, and read the description of the provided helpers at the end of this page.

Semantic errors are reported by returning an error message string instead of `None` from the functions provided to the visitor implementation. The provided visitor then prints out this error message and stops the program. If syntax tree nodes contain an attribute called "`lineno`", the line number is included in the error message.

Running the program

After running semantic checks, you now should have a syntax tree (program) which can be executed. We suggest that you write an interpreter which interprets (= runs) the syntax tree. (Alternatively you can generate code from the tree, i.e. write a compiler, but that's probably more complicated than interpretation. If you choose this option, contact the course staff first).

While running the program you don't have to implement any run-time semantic checks, if you don't want to. I.e. you can assume the program is semantically correct, even if you don't write all semantic checks, and you can assume there are no run-time errors which would require error handling in the interpreter.

Interpreting

One way to implement an interpreter is to have an "eval()" method that recursively evaluates the nodes in the syntax tree. You call the topmost eval() and it goes through the tree executing the node statements in the correct order. E.g. before running a plus(+) operation the left and right parts are evaluated first (the value of the evaluated expression can be the return value of eval()). The example Unicode calculator can be used as the basis for your interpreter (or you are welcome to design your own).

Levels of implementation

Below are different levels of implementing execution. Since levels depend on each other, it's advisable to implement them in order. Also, it's ok if some level is only implemented partially (of course that will be taken into account in grading).

1. Evaluation of expressions consisting of arithmetic expressions and *integer* literals. The interpret should print out the value returned from the top-level return statement "`=`".
2. Additionally, using integer variables works (i.e., definition and reading the value).
3. Evaluation tuple variables and tuple expressions (including "`select:`" but not including pipes).
4. In addition to above, function definitions, function calls and parameter passing works.
5. Evaluation of pipe operations.
6. Implementing recursion and proper local variables and parameters.

Grading in this phase

From this phase you can raise your preliminary grade from phase 3 by at most 3 grades:

- If you implement at least 2 semantic checks, 2 level of implementation, or 1 level of both, you can improve your grade by 1.
- If you implement at least 4 things that include both semantic checks and implementation levels, you can improve your grade by 2 (i.e., doing only semantic checks or only implementation levels is not enough).
- If you implement more than 5 things, your chances of raising the grade by 3 points improves. Remember that it's possible to get grade 6 from this project, i.e. "really excellent"!
- If you come up with semantic checks/implementation things of your own, those can help improving the grade, provided that you also document them well.
- **NOTE:** The above is the maximum grade improvement, assuming that your semantic check / implementation level is "well done". It's of course possible to get partial points for incomplete / less well done things.

Minimal functionality

Your program must start when executed in python by: `python3 main.py` in your submission directory. You can assume that this python interpreter has PLY installed.

Your program must understand the following command line arguments:

```
usage: main.py [-h] [--who | -f FILE]
  -h, --help            show this help message and exit
```

```
  --who                 print out student IDs and NAMEs of authors
  -f FILE, --file FILE  filename to process
```

(See the course snippet on how to implement this using python standard library.)

Errors

Your code should be built on top the previous phase (syntax), so the same errors messages from there should be used here also. As mentioned before, you don't have to implement run-time error checking, if writing the interpreter.

Required text document

In addition to the code, the group submits a text document in either plain text (.txt), markdown (.md) or pdf (.pdf) format. It should document each semantic check / implementation level you've done, explaining how it works. If you implemented any things of your own, explain them also in the document. Also tell what you thought about this assignment? What was difficult? What was easy? Did you learn anything useful?

Extras

If you have an idea, feel free to implement it (AND remember to document it)

Tools provided by the course

The file semantics_common.py contains a helper function visit_tree() and classes SemData and SymbolData to help with semantic checks (the classes may be useful also during interpretation).

- The `SemData` class is meant to be used as a place where you can store all kinds of information required by the semantic checks and interpretation, including the symbol table (which is already included in the class). Create a single object of this class at the beginning, and pass it to every function as a parameter to share data across the checking the interpretation. Examples of possible stuff to keep in SemData (in addition to the symbol table) are call stack (interpretation), what function you are in (semantic checking), etc.
- The `SymbolData` class is an example of what you could use as a symbol table entry (i.e. store in the .symtbl of SemData). Now it contains the type of the symbol (variable, function, you choose what to put there) and a pointer to the node where the symbol is defined in the tree, but you can choose to put anything there.
- Finally, the `visit_tree` function can be used during semantic checking to perform operations on every node of the tree. You give it two functions: one function is called when `visit_tree` starts handling a node, and the other when all the children of the node have been handled (visited). In those functions you can choose which nodes you are interested in, collect information to `SemData` or the node itself, and/or check that things are semantically ok.

Questions with answers

Nothing at the moment.

General practical questions about the course should be sent to popl@lists.tuni.fi.