University of Liège
Faculty of applied sciences

Project report:

Development of a compiler for a new programming language

INFO0085-1: Compilers

CATUSANU Paul - EWBANK Tom - VAN DE GOOR Elodie
s?????? - s102705 - s??????
1er Master Ing. Civ. Inf.

Contents

1 Our language: Raccoon

We wanted to have a language easy to understand and used as first language. Our language definition was driven by the python philosophy of “Simple is better than complex” and “Beautiful is better than ugly”. So, we inspired ourselves of the python syntax for much of our language, particularly the indentation to structure the code instead of the punctuation, in order to improve readability.

Since our goal is to make a language easy to use for a child who has never programmed before, the Raccoon will have two key particularities:

- Use of meaningful keywords instead of some usual operators
- Dynamic typing

The features of the Raccoon language are detailed in the following sections.

1.1 Types

There are five types in Raccoon: Double, Integer, Boolean, List and String.

Numeric types: All the arithmetic operations, the combinatorial operations and the comparison operations are allowed with those types. When an operation involves different numeric types, the resulting type is considered as Double. Boolean are considered as numeric types, with a value of 1 for "True" and 0 for "False".

Lists: All the elements of a list must have the same type category (Integer, Double and Boolean are part of the same category, numeric types). The expression to define a list has the form "[*element1, element2,...,elementN*]". The only authorized operations on a list are the access and the modification of its elements. The expression to access or modify an element has the form "*listName*[*index*]" where *index* is an Integer (or an expression leading to one. The index of a list starts at 1. A list is also not allowed to contain other lists.

Strings: No operations are allowed on Strings. They can only be instantiated and displayed. The expression of a string consist in text surrounded by quotes.

1.2 Assignments

An assignment has the form:

variableName **becomes** *expression*

1.3 Constants

The definition of a constant has the form:

constantName **is** *expression*

A constant can be of any type.

1.4 Operations

Raccoon supports the usual operators:

Arithmetic operators: +, -, *, /, mod.

Combinatorial operators: and, or.

Comparison operators: =?, <, <=, >, >=.

1.5 Indentation

The indentation has to be made of tabulations only, not spaces.

1.6 Function definition

The definition of a function has the form:

function *name*(*argument1, argument2,...,argumentN*)**:**

instruction1
instruction2
...
instructionN

A function has to contain at least one instruction, and of course, the use of a "**return** *expression*" statement is allowed.

Functions doesn't need to be declared before being defined.

Arguments are passed by values.

Polymorphism is not allowed in Raccoon. It simplifies things and, as this language is intended for young people with no experience in programming, and thus with no intention of building very complex programs, we think that this limitation is not really problematic.

1.7 Conditionnals

A conditionnal statement has the form:

if *condition* **:**
    instruction1
    instruction2
    ...
    instructionN
else if *condition* **:**
    instruction1
    instruction2
    ...
    instructionN
else:
    instruction1
    instruction2
    ...
    instructionN

The "**else if**" and "**else**" blocks are not mandatory and having more than one "**else if**" in the same conditionnal is allowed.

1.8 Loops

Raccoon provides "while" and "for" loops, along with the special instructions "**break**" and "**continue**".

1.8.1 "while" loop

This loop statement has the form:

while *condition* **:**
    instruction1
    instruction2
    ...
    instructionN

1.8.2 "for" loop

This loop statement has two possible forms:

for *iterator* **in range(***start,end***):**
instruction1
instruction2
...
instructionN

and

for *iterator* **in** *listName***:**
instruction1
instruction2
...
instructionN

They work the same as in Python.

1.9 Scope of variables

The scope of the variables is defined by the function definitions. Each time you enter a new function definition, you enter a new scope. Variables are only accessible in the scope where they were initialized or defined as arguments.

However, there is also a particular scope for the variables used as iterator in a "**for**" loop. The scope of these variables is limited to the "**for**" block. If a variable was already defined with the same name, this variable is not accessible in the scope of the "**for**", every uses of this variable inside the "**for**" statement will refer to the iterator.

1.10 Comments

Line comments starts with "//" and block comments are delimited by "/*" and "*/".

2 The compiler

2.1 Implementation language and tools used

The compiler is written in Python. What motivated us to choose this language is the fact that we found very helpful tools implemented in Python. We have indeed found the website of some informatics teacher named Matthieu Amiguet, who provides a well explained tutorial about the development of a simple compiler, as well as a very handy module which permits to represent and manipulate abstract syntax trees.

The main tool we used in this project, as it is used in Mr. Amiguet's tutorial, is a module called PLY, which is a pure-Python implementation of the popular compiler construction tools lex and yacc.

2.2 Development

The implementation of our compiler is separated into five files. One of these files is a modified version of Mr. Amiguet's module "AST.py" for the creation and the handling of abstract syntax trees. The other four files are each responsible for a different step of the compilation process.

2.2.1 Lexical analysis

The lexical analysis is implemented in the file "lexical.py".

This analysis is divided in two stages. The first stage is aimed at generating a temporary list of tokens, which will then be modified during the second stage in order to take into account the indentation present in the program file that's being analysed.

The first lexing stage: This stage simply uses the lex functionality of PLY to generate the list of tokens found in the input file. Among these tokens, the ones called "END_STATEMENT" contain the informations regarding the indentation. The reason of this is that the pattern for this token matches any new line, along with eventual empty lines and tabulations that follow.

The second lexing stage: The role of this stage is to represent the indentation in the token list, using special "INDENT" and "DEDENT" tokens. An occurence of "INDENT" in the list means that every new lines that follows are shifted by a fixed number of tabulations. "DEDENT" cancel the effect of the previous "INDENT" while a new occurence of "INDENT" adds a new shift compared to the previous one. That being said, what this stage do is going through the token list to find the tokens

"END_STATEMENT", deducing the indentation from the tabulations stored in it, and, when necessary, adding new tokens of type "INDENT" and "DEDENT" to the list. The principle of the deduction process is not very complicated, it just requires to keep track of the previous levels of indentation and compare them to the new one.

However, with this approach, we coudn't think of a way to include the tokenization of the comments. We thus found a little trick to solve this problem. Instead of passing the original input string to our lexical analyser, we pass a modified version, where all the comments have been removed. We are nonetheless aware that this method is not the most efficient since it requires to go through the whole file one more time.

There is still a detail that was needed to make the second lexing stage works. In order to be able to generate the last "END_STATEMENT" and the right number of "DEDENT" at the end of the token list, the program must end with a "\n" character. So, instead of forcing the programmer to end his programs with a new empty line, our compiler just adds it automatically to the input string before starting the analyse.

Finally, it is worth mentioning that some error reporting is done during this phase. The lexical analyser will report an error for each illegal character encountered, and for each erroneous level of indentation. By erroneous, we mean that the level of the indentation has decreased, but doesn't correspond to any level of indentation previously encountered. Let's take an example to make things clearer:

```
1 function example():
2                     a_fct_call()
3         display(something)
```

In that case, we got an error at line 3, which should be align either with line 2, either with line 1, but not create a new indentation level between the two. Note that line 3 could also be switched to the right compared to line 2, it is perfectly correct at this stage, but won't be when checking the syntax. The analyser also warns the user if it detects the use of spaces that could corrupt the indentation, but those warnings won't be considered as actual errors, and so they won't prevent the program from being compiled, but it is not a problem, because if the spaces are really corrupting the indentation, the parser will notice it. Those warnings are thus there to provide some additional information to the error reporting done by the parser, but that topic will be discussed in the next section.

2.2.2 Parsing

The parsing is implemented in the file "parsing.py".

The parsing component of PLY is based on Yacc, and Yacc uses the parsing technique known as LR-parsing or shift-reduce parsing. LR parsing is a bottom up

technique that tries to recognize the right-hand-side of various grammar rules. Our grammar rules are stated in the docstrings of a series of functions and whenever a valid right-hand-side is found in the input, the code inside the corresponding function is executed and the grammar symbols are replaced by the grammar symbol on the left-hand-side.

The grammar, as we implemented it here, is ambiguous. However, the parser can deal with it and inform the user of the actions taken, in the form of 2 messages: "shift/reduce conflicts" or "reduce/reduce conflicts". A shift/reduce conflict is caused when the parser generator can't decide whether or not to reduce a rule or shift a symbol on the parsing stack, and by default, all shift/reduce conflicts are resolved in favor of shifting. In our case, as a matter of fact, this default action solves our ambiguity problems, except for arithmetic or comparison expression rules, which we wrote in this very simple way:

```
'''expr : expr ADD_OP expr
        | expr SUB_OP expr
        | expr MUL_OP expr
        | expr DIV_OP expr
        | expr MOD_OP expr
        | expr comb_op expr
        | expr comp_op expr'''
```

But once again, the Yacc module of PLY provides a way to deal with it. Indeed, it allowed us to define precedence rules for our arithmetic/comparison operators, so that the ambiguity is gone.

Of course, this parser is not only responsible for checking the syntax of a Raccoon program, it is also responsible for the building of an abstract syntax tree which will be used during the next steps of the compilation process. That's where the series of functions containing the grammar rules are involved. These functions permits to creates the different nodes of the tree and link them together, according to the rules to which they correspond. As said before, the data structure of the tree is implemented in the module "AST.py", originally written by Matthieu Amiguet, but specifically adapted to be used with the Raccoon language. The modifications we applied at this stage was the implementation of several subclasses of the basic "Node" class. These subclasses are the different types of node that can compose the abstract syntax tree. As an illustration, the abstract syntax tree generated by our implementation of the QuickSort is shown in figure 1.

The last point to address concerning this part is the error reporting. The ideal way of reporting errors would be to have grammar rules for each possible errors, so that the compiler could precisely tell the user what he did wrong. However, that implies that we should think of every possible error cases. So, in order to avoid overloading our grammar with additional rules, and risking leaving bugs in our compiler, we implemented an hybrid solution, which consist in having some grammar rules for the most common errors, and also using what's called a "panic mode recovery".

Whenever the panic mode recovery encounters an unknown error, it will first report it, then discard it and restart the parsing from the next token. Because of that, the parser may generate additional errors. So, this is not the perfect solution but, since the Raccoon grammar is not too complicated, we thought that it was sufficient to help the user correcting his mistakes.

2.2.3 Semantic analysis

The semantic analysis is implemented in the file "semantic.py". In this module, functions are added to the nodes of the syntax tree, thanks to a decorator.

The first part of these functions permits the "couture" of the AST, allowing a particular traversing of the tree. To sew a node, we first need to sew recursively its children, then the node itself. So, we provided a method to the Node class, that takes as argument the last sewed node, sews the corresponding sub-tree and returns the new "last sewed node". In addition to that, we had to provide a function to start the couture process. This function creates an instance of a special node type, called "EntryNode", which defines the entry point of the program. An example of a sewed tree is given at figure 2, the "couture" is represented by the coloured links.

Once the tree is sewed, we can follow this path and do some semantic checking. This work is done by adding a method "semAnalysis" to each type of node. Each node class has thus its own implementation of this method, because different actions has to be performed depending on the type of node. The only common point to all the nodes is the last instruction of the method, which is of course a call to "semAnalysis" on the next node in the couture.

This analysis performs some checking about the following:

- **Variables initializations** : each time time an assignment node is encountered, it creates (or modify) an entry in a hash-table. This entry correspond to the name of the variable to which we assign a value, and is associated with the type being assigned. The same happens for the arguments of a function definition, except that this time the types are unknown. That way, each time a variable is used somewhere, we can look up in the hash-table if it was well initialized.
- **Functions definitions** : This works the same way as with variables initialisation, with the difference that the information stored along with the name of the function is here the number of arguments. We can then verify if a function call is correct.
- **Types** : Although our language is based on implicit typing, it is still possible to do some type-checking. Types can be sometimes deduced, and when they can't, we consider them as unknown types which won't interfere with the analysis. The principle here is to keep a stack with the encountered types, and sometimes merge the last two types on the top of stack, for example when

an operation node is encountered. This results sometimes in a "forbidden" type, as long as an error report, in case the types are not compatible. This system allows us to report several sorts of erroneous type uses like mixing numeric types with strings, creating list where the elements are not all of the same type, trying to access list elements with something else than an integer,...

- **Non context-free statements** : "break" and "continue" instructions has to be used inside a loop, while the "return" statement has to be inside a function definition. This is verified by going trough the parents of the node until it reaches either a loop node or a function definition node, depending on the case. If it reaches the root of the tree before that, it means that the statement is not at the right place.

In order to make all these checkings possible, we obviously needed to implement some data structures aimed at containing the type-stack, the hash-tables but also aimed at keeping track of the different scopes of the program. We thus created two new classes in the AST module. The first one represents a scope of the program and contains the hash-tables and the type-stack associated to this scope. The second one maintains a stack whose elements are instances of the previous class. The scope on the top of the stack is the one being manipulated during the analysis. So, each time the semantic analysis enters a new scope, a new "Scope" object is pushed on the stack. Whenever it leaves a scope, it simply pops the last object that was pushed on stack, getting back to the previous scope. This class which handle the stack of scopes is named "ScopeStack" and we instantiates one as a class variable of the "Node" class so that it will be shared by each nodes of the tree.

2.2.4 Code generation

2.3 How to use

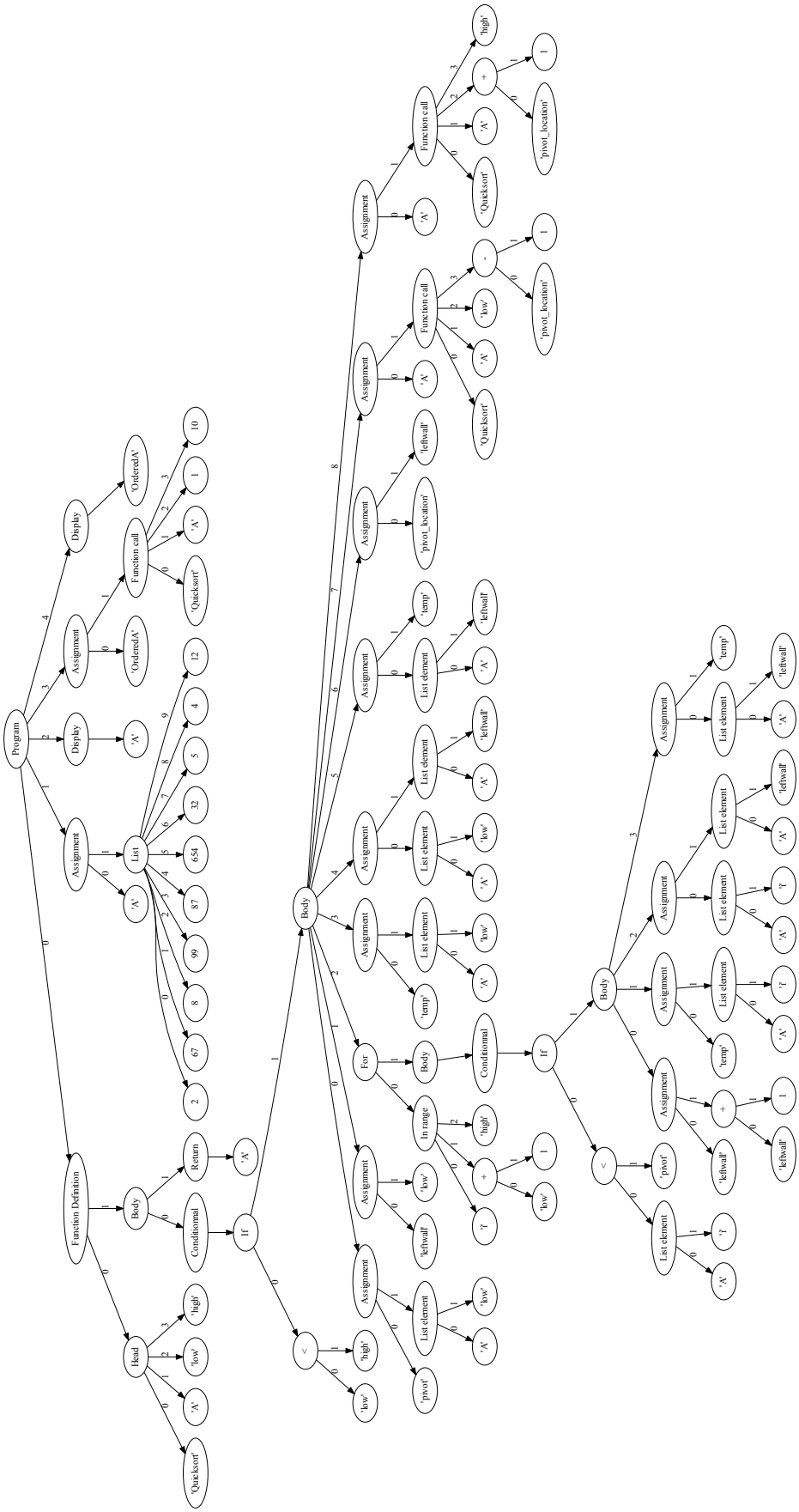

Figure 1: AST of the QuickSort program

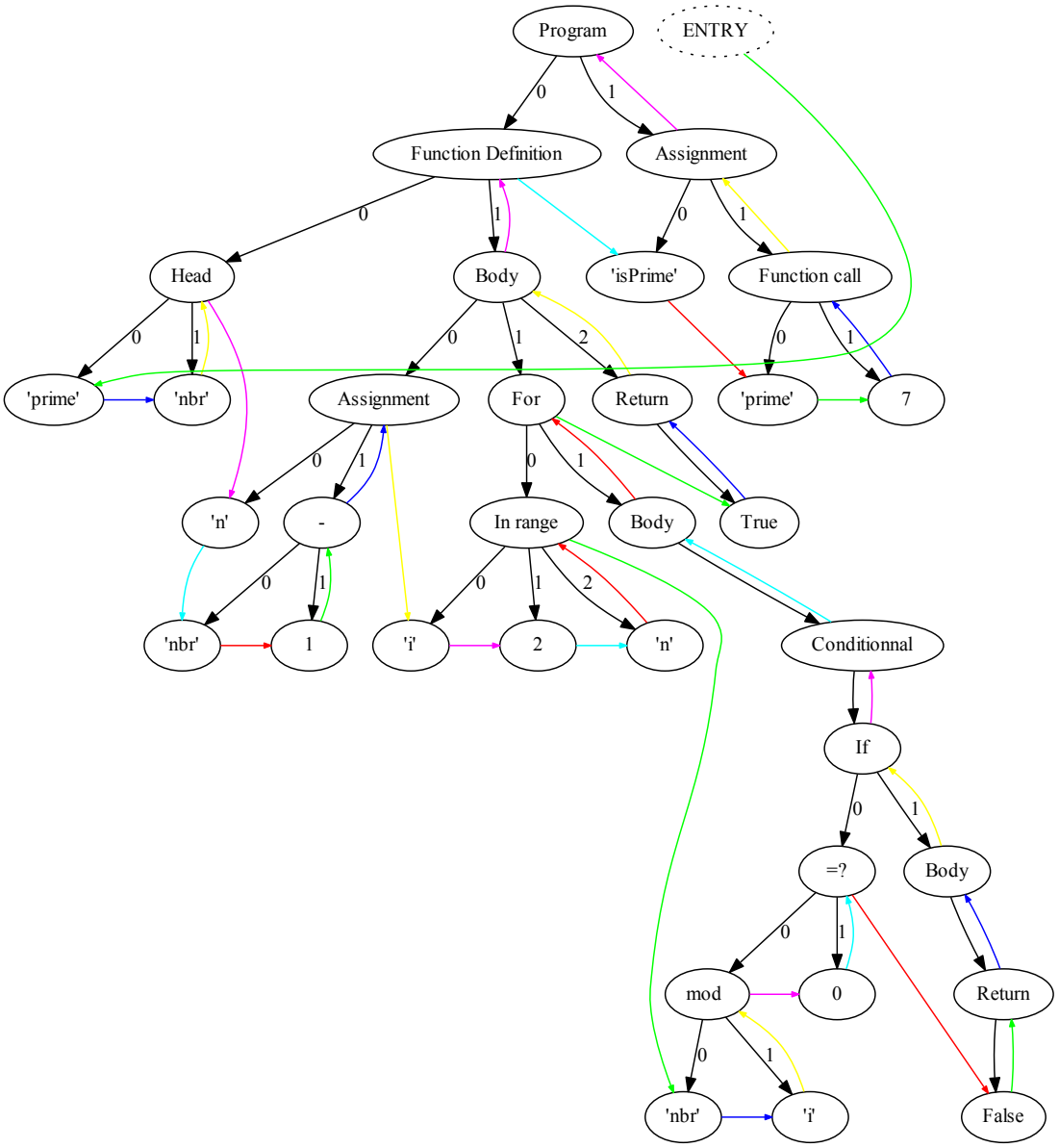

Figure 2: Couture of an AST

References

[1] http://www.montefiore.ulg.ac.be/~geurts/Cours/compil/2014/compilers-geurts-february2015.pdf

[2] http://www.matthieuamiguet.ch/pages/compilateurs

[3] http://www.dabeaz.com/ply/ply.html