ChAmElEoN Parse Tree

Jack L. Watkin

May 9, 2017

The objective of this appendix is to describe the abstract syntax tree (AST) generated by the ChAmElEoN parser.

1 Tree Node

The ChAmElEoNparser builds an AST in which each node contains the node type, a leaf, a list of children, and a line number. The tree node structure is shown in Fig. 1.

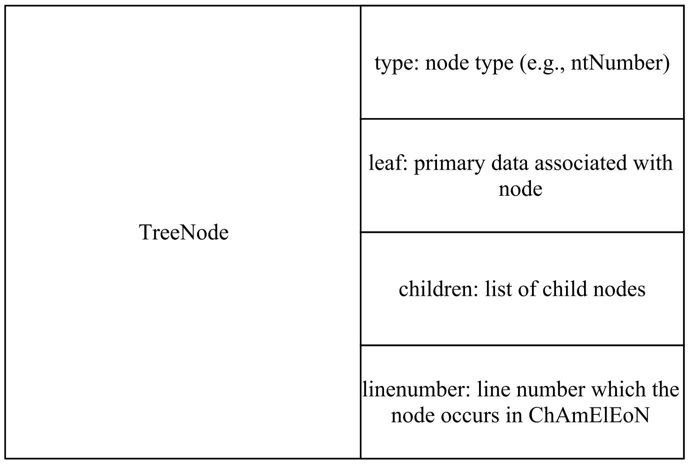
TreeNode
type: node type (e.g., ntNumber)
leaf: primary data associated with node
children: list of child nodes
linenumber: line number which the node occurs in ChAmElEoN

Figure 1: Visual representation of `TreeNode class`.

2 Numbers, Identifiers, and Operators

For all numbers (`ntNumber`), identifiers (`ntIdentifier`), and primitive operators (`ntPrimitive`) the value of the token is stored in the leaf of the node (e.g., Fig 2).

3 Interpreter Overview

The `evaluate_expr` function is the main loop of the interpreter. Each node type corresponds to a case of the conditional statement in `evaluate_expr`. The following shows a simplified example `evaluate_expr` of the function.

```
def evaluate_expr(expr, environ):
    if expr.type == ntPrimative_op:
        ...
    elif expr.type == ntNumber:
        ...
```

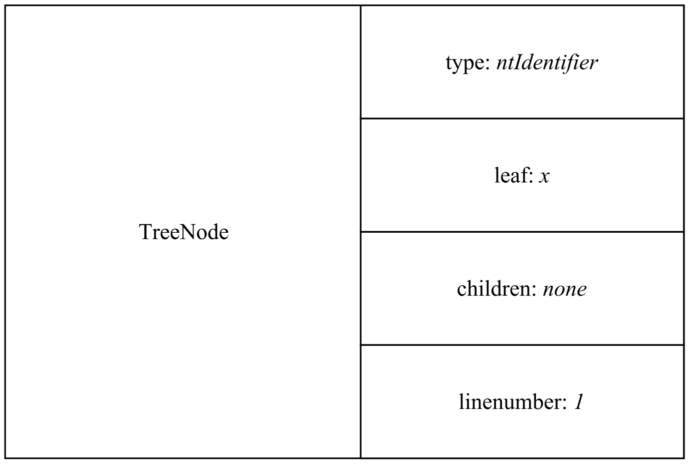

Figure 2: Example of a `TreeNode` for an identifier.

```
    elif expr.type == ntIdentifier:
        ...
```

4 Arguments and Expression Lists

4.1 Abstract Syntax Tree

The following rules are used to represent an argument list. Argument lists are used to describe the arguments to a function or primitive.

$$\begin{array}{lllll} (r_1) & <argument\_list> & ::= & <argument> \\ (r_2) & <argument\_list> & ::= & <argument>, <argument\_list> \end{array}$$

Similarly, the following rules are used to represent a parameter list. Parameter lists are used to describe the parameters in a function declaration.

$$\begin{array}{llll} (r_1) & <parameter\_list> & ::= & <identifier> \\ (r_2) & <parameter\_list> & ::= & <identifier>, <parameter\_list> \\ (r_3) & <parameter\_list> & ::= & \epsilon \end{array}$$

Argument and parameter lists are built by the parser as trees and then transmuted into a list a by the interpreter. Fig. 3 shows the AST that describes the expression `*(7,x)`. A blue box represents the `type` field of the `TreeNode`, a red box represents represents one member of the `children` list, and a green box represents the `leaf` field. Note that expression is an overloaded operator in that it is used with both the arguments list to a primitive and as the arguments to a function.

4.2 Interpreter

Both argument and parameter list ASTs are traversed such that the right recursive case is performed last. The leaf of the parameter/argument list node is placed at the front of a empty list. In argument lists, value of the leaf is evaluated before placing it in the list. If a child exists, it becomes the next parameter/argument list node to be searched. This continues until an parameter/argument list node without a child is reached. A list is returned by the recursive case and appended to list the created with the leaf of the node. The following is an example from the interpreter that illustrates this idea.

```
elif expr.type == ntArgumentList:
    ArgList = []
```

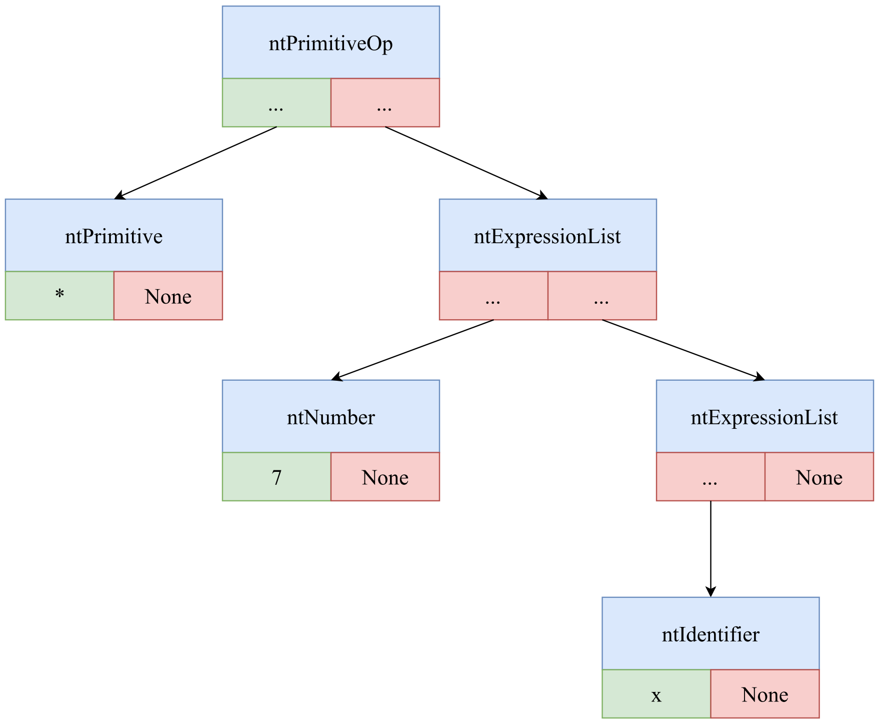

Figure 3: Example of tree containing an argument list.

```
    ArgList.append (evaluate_expr(expr.children[0], environ))

    if len ( expr.children ) > 1:
        ArgList.extend (evaluate_expr(expr.children[1], environ))
    return ArgList
```

5 let, let*, and letrec

5.1 Abstract Syntax Tree

The AST that describes a `let`, `let*`, or `letrec` expression is similar to the AST which describes an argument list. These statements rely on a right recursive grammatical structure to produce the desired AST. One nuance in this grammar is that this does not allow for `let` statements with no declarations.

$$\begin{array}{lrcl}
(r_1) & \langle expression\rangle & ::= & \texttt{let}\ \langle let\_statement\rangle\ \texttt{in}\ \langle expression\rangle \\
(r_2) & \langle let\_statement\rangle & ::= & \langle let\_assignment\rangle \\
(r_3) & \langle let\_statement\rangle & ::= & \langle let\_assignment\rangle\ \langle let\_statement\rangle \\
(r_4) & \langle let\_assignment\rangle & ::= & \langle identifier\rangle = \langle expression\rangle
\end{array}$$

The `let*` and `letrec` expressions are defined with similar rules with an exception in `letrec`. With respect to *<letrec_assignment>* rule, only a function can be assigned to the identifier. *<recursive_func>* can be used to build an AST for a recursive or non-recursive function. This is done because the interpreter needs to know whether to build the function with itself in the environment or not.

$$(r_1) \quad \langle letrec\_assignment\rangle \quad ::= \quad \langle identifier\rangle = \langle recursive\_func\rangle$$

Fig. 4 shows a simplified version of the AST that represents a `let` statement. This is also applicable for `letrec` and `let*`.

```
let
    x = 1
    y = 2
in
    *(x,y)
```

As before, a blue box represents the `type` field of the `TreeNode`, a red box represents represents one member of the `children` list and a green box represents the `leaf` field.

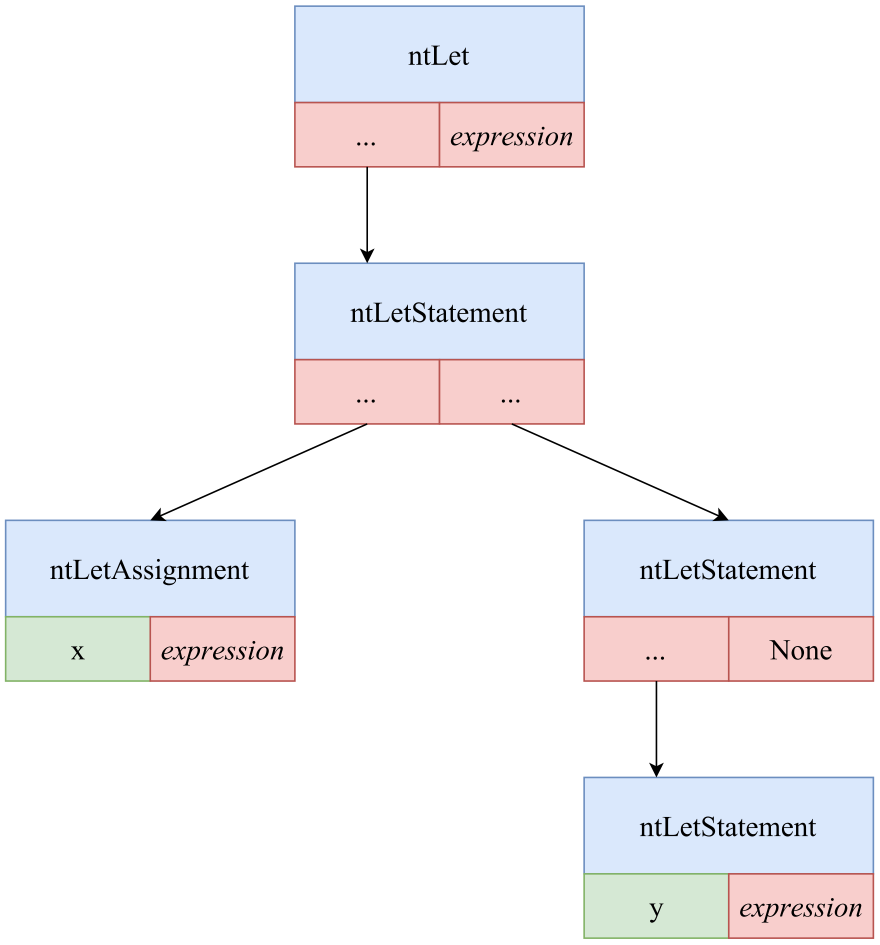

Figure 4: Example of a tree containing a `let` statement.

5.2 Interpreter

While the AST for `let`, `let*`, or `letrec` are largely the same, the differences in their functionality arises from how the interpreter processes the tree. Similar to parameter/argument lists, the tree that describes a `let` is searched in a preorder fashion from within the `ntLet` case. The `ntLetAssignment` case creates a single element dictionary containing a name value pair defined within the `let`.

```
elif (expr.type == ntLetStatement):
    temp = evaluate_expr (expr.children[0], environ)
    if len ( expr.children ) > 1:
        temp.update (evaluate_expr(expr.children[1], environ))
    return temp
```

```
elif expr.type == ntLetAssignment:
    return {expr.leaf : evaluate_expr(expr.children[0],environ)}
```

When all `ntLetStatements` have been processed, a dictionary containing all name value pairs is returned to the `ntLet` case. The dictionary is then split into two list—one containing only names, the other containing only values. These values are placed into the environment. The body of the `let` statement is called with this new environment.

The `ntLetrecStatement` and `ntLetstarStatement` traversed in the same way, however these statements differ in how values are added to the environment. For `let*` statements, each declaration inside the statement is added one by one. This allows for values declared at the beginning of the `let*` to be used within subsequent statements in the `let*`. This stands in contrast to how the `let` statement adds them all at once. For `letrec` statements, the difference with `let` lies only in that the `ntLetRecAssignment` returns a list containing three lists: one that is a list of identifier to which each function is bound, the argument lists of each function, and the body of each function.

6 Functions

In ChAmElEoN, functions are represented as closures in one of three forms: abstract syntax representation (ASR), closure representation (CLS), or Python closure representation. For all representations, the arguments For ASR and CLS representations, the environment the function is declared in is stored in the closure (static scoping). Fig. 5 shows the ASR closure. When invoked, these closures are passed the values to be bound to the arguments of a function. Recursive functions are created by storing a pointer to the function within the environment, and then creating a closure.

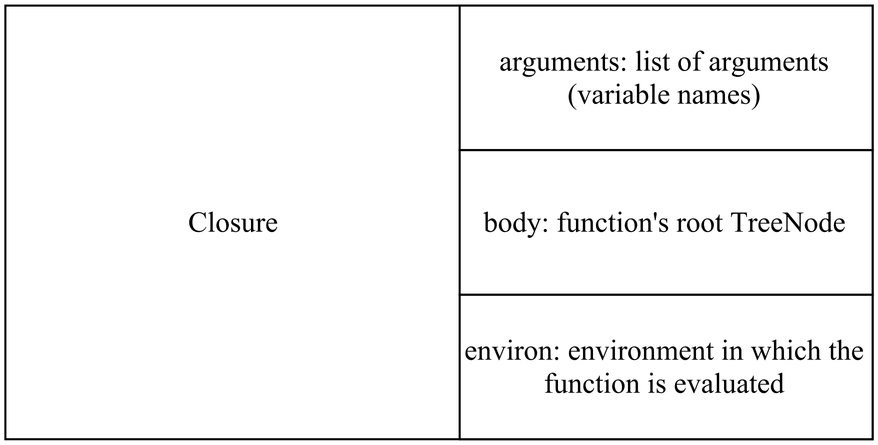
Closure
arguments: list of arguments (variable names)
body: function's root TreeNode
environ: environment in which the function is evaluated

Figure 5: Abstract syntax representation of a closure.

For Python closures, the environment is passed to the closure when the function is called (dynamic scoping).

7 Debugging

The way that exceptions are handled within this interpreter causes issues for debugging. There are several catch-all statements in the interpreter that serve to trap errors, but do not present helpful debugging messages. For example, an unbound identifier exception does not necessarily mean that there is an unbound identifier. Rather, it means there was an unhandled exception is attempting to resolve the identifiers value.

When debugging the environment, list of vector representation is the easiest print. This is because the environment is a list, which can be printed with the Python `print` function. For abstract syntax representation, use `environ.symbols` and `environ.values`. Closure representations of the environment cannot be easily printed outside the closure function itself.