Closure Compiler AST Documentation

Nick Santos

September 22, 2010

1 Objective

The purpose of this document is to describe how Closure Compiler represents JavaScript source code internally. Our intended audience is people who want to add new compiler passes.

In order to ensure this document is up to date, most of it is auto-generated from the Closure Compiler source.

2 Background

Closure Compiler has 3 different abstract syntax trees (abbreviated as ASTs).

When Closure Compiler started in 2004, it used Rhino (`http://www.mozilla.org/rhino/`) to parse the JavaScript into an AST, performed checks and optimizations by directly modifying that tree in-place, and then serialized the tree back out to JavaScript source.

Over time, Closure Compiler's AST diverged from Rhino's AST. Some of this divergence is cosmetic ("you say tom-ay-to, and I say tom-ah-to"). But there are also some new features. Keep in mind that Rhino's AST is designed to make it easy to execute the code, whereas Closure Compiler is more interested in static analysis. Differences between the two ASTs include:

- Closure Compiler includes a JSDoc comment parser that parses JSDoc annotations, and then attaches those annotations to nodes. The code that attaches the annotations

to nodes has been contributed back to Rhino, but the JSDoc comment parser lives in `com.google.javascript.jscomp.parsing`.

- Closure Compiler attaches type information to most AST nodes when type-checking is turned on.
- Rhino and Closure Compiler have both added control flow graphs, but Closure's control flow graph is more decoupled from the AST (it sits on top of the AST instead of inside it).

Closure Compiler still uses Rhino to parse JavaScript source code. First, we feed the files to Rhino's parser. Then, the classes in `com.google.javascript.jscomp.parsing` know how to transform Rhino's AST into Closure Compiler's AST. Occasionally, we make changes to the AST to make the code easier to analyze.

Over time, we realized that it would be even easier to do good optimizations if equivalent statements were represented in a canonical way. For example, `var x,y;` and `var x; var y;` are equivalent, even though they are syntactically different. JavaScript allows a lot of syntactic sugar to make it easier to do things in shorter ways.

Before Closure Compiler runs its optimization passes, it transforms the original AST into a normalized AST. The normalized AST is exactly what it sounds like: an AST where syntactic sugar has been transformed into a canonical form. For example, variable declaration statements with multiple names (`var x,y;`) are split into multiple statements (`var x; var y;`). By definition, the set of valid normalized trees are a strict subset of the complete set of valid ASTs. The structure of the normalized AST does not have to match the syntactic structure of the original code. This normalizer lives in `com.google.javascript.jscomp.Normalize`. This transformation is much simpler than the Rhino transformation, and leaves most of the tree as-is.

To make it easier to debug and develop against these ASTs, Closure Compiler has a tree debugger that serializes the final state of the AST into a DOT file, the file format that Graphviz (`http://www.graphviz.org/`) uses to represent graph structures. There are many publicly available tools for rendering these graphs as images.

3 Methodology

By default, Closure Compiler reads JavaScript source from `stdin`. On a UNIX-style shell, invoking the command:

```
echo ''/** comment */ var x = 3;'' | java -jar compiler.jar
```

will parse the echoed code into an AST, then serialize it to compressed JavaScript:

```
var x=3;
```

If we pass the `--print_ast` option to the compiler, we will instead get the graph of the optimized tree in DOT form. Notice that this is the AST of the *output* code, not the input code.

```
echo ''/** comment */ var x = 3;'' | java -jar compiler.jar --print_ast true
--jscomp_warning checkTypes
```

```
digraph AST {
  node [color=lightblue2, style=filled];
  node0 [label="BLOCK"];
  node1 [label="SCRIPT"];
  node0 -> node1 [weight=1];
  node2 [label="VAR" color="green"];
  node1 -> node2 [weight=1];
  node3 [label="NAME : number"];
  node2 -> node3 [weight=1];
  node4 [label="NUMBER : number"];
  node3 -> node4 [weight=1];
  node2 -> RETURN [label="UNCOND", fontcolor="red", weight=0.01, color="red"];
  node1 -> node2 [label="UNCOND", fontcolor="red", weight=0.01, color="red"];
  node0 -> RETURN [label="SYN_BLOCK", fontcolor="red", weight=0.01, color="red"]
;
  node0 -> node1 [label="UNCOND", fontcolor="red", weight=0.01, color="red"];
}
```

If we feed this to the `dot` tool, we get a pretty graph.

Each node of graph may contain:

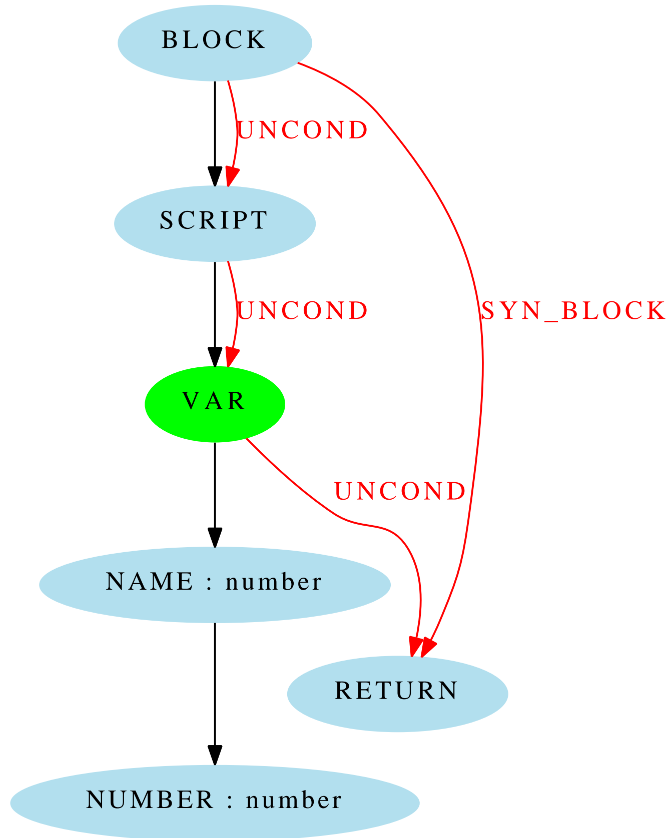


Figure 1: A sample AST.

- The name of the syntactic token that the node represents.
- The type information attached to the node.
- Black edges to the nodes children and direct parent.
- Red edges that represent the control flow graph.
- A solid green fill if the node has a JSDoc comment attached.

4 Syntax Trees

This section contains syntax trees that have been auto-generated.

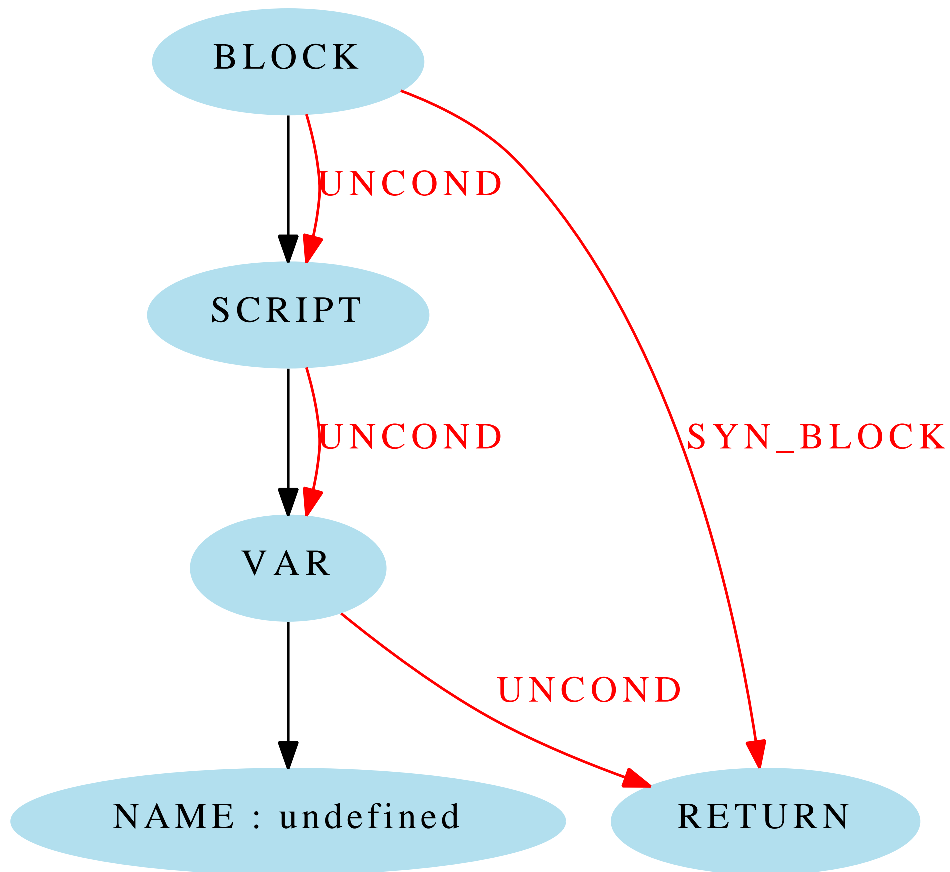


Figure 2: var x;

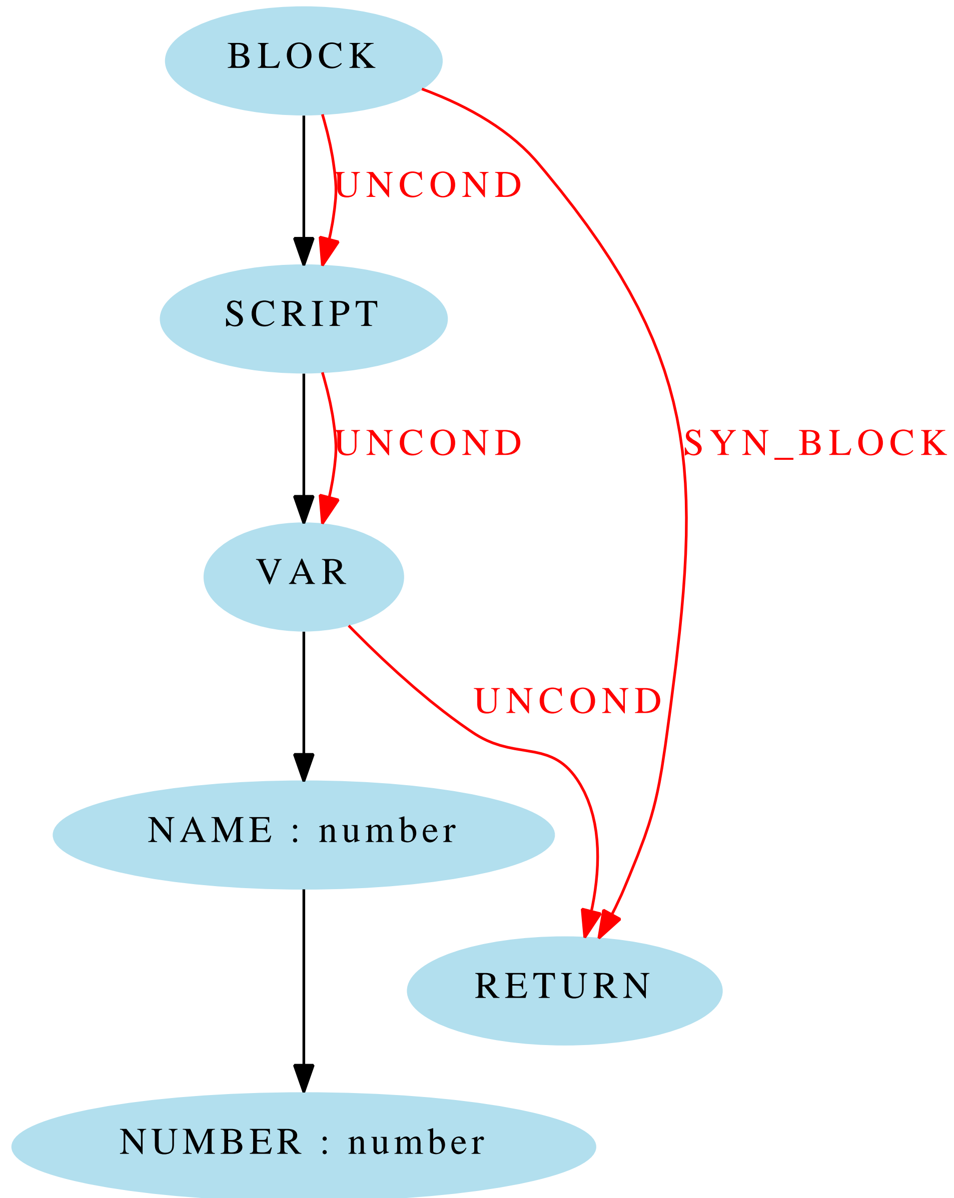


Figure 3: var x = 3;

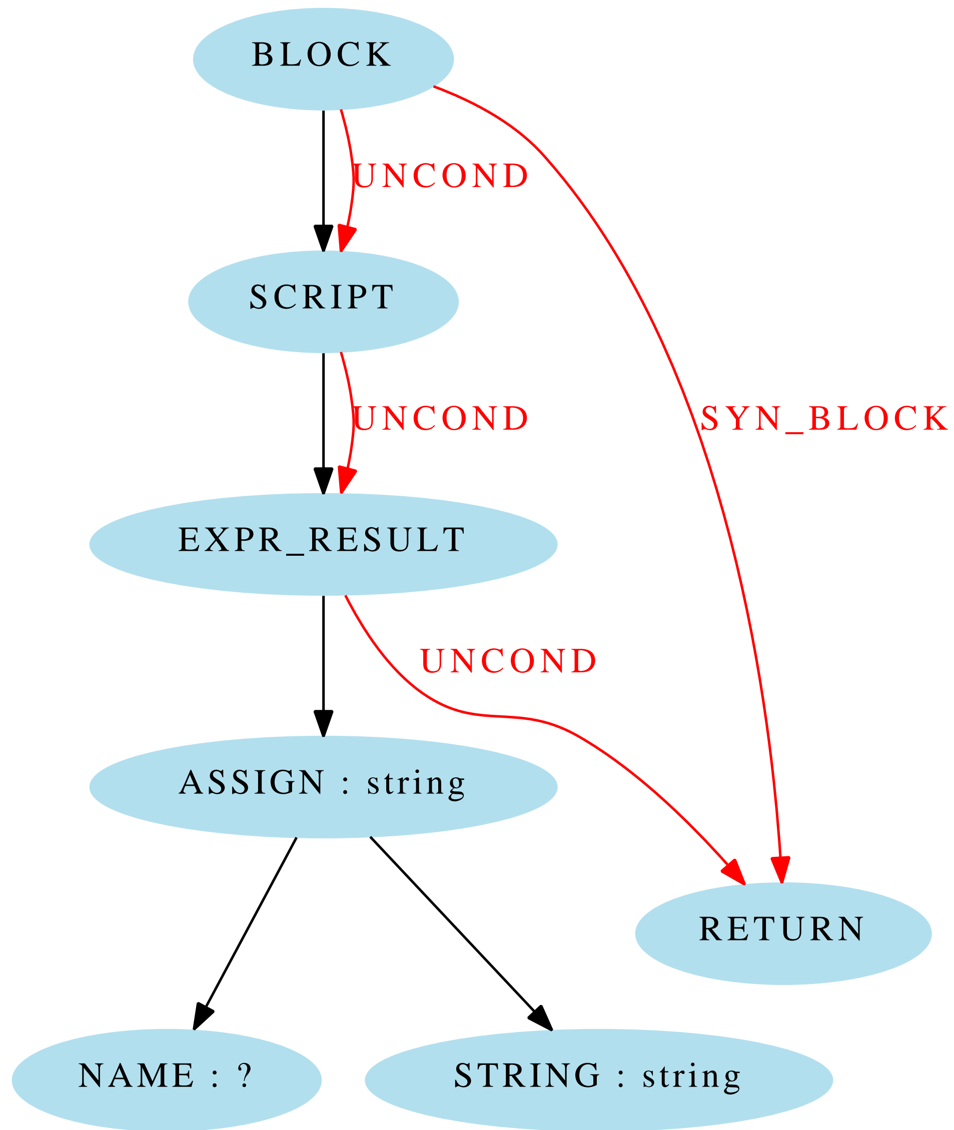


Figure 4: x = 'string';

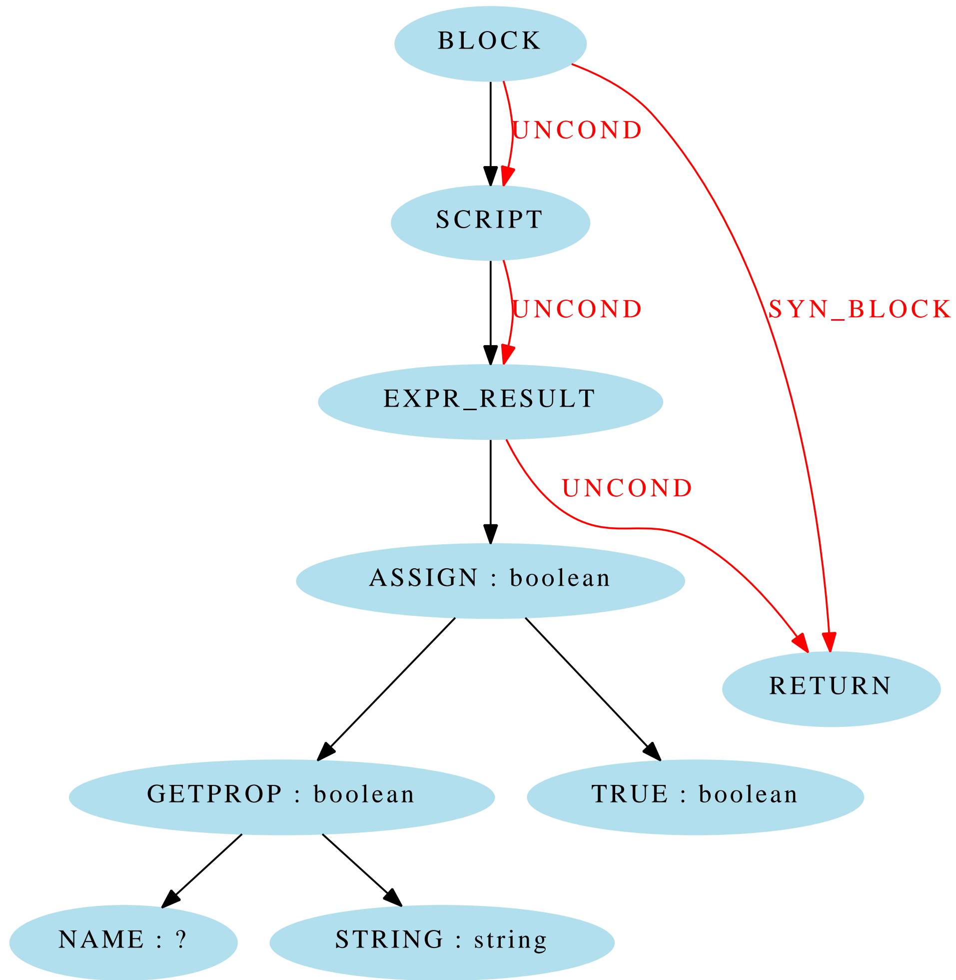


Figure 5: x.y = true;

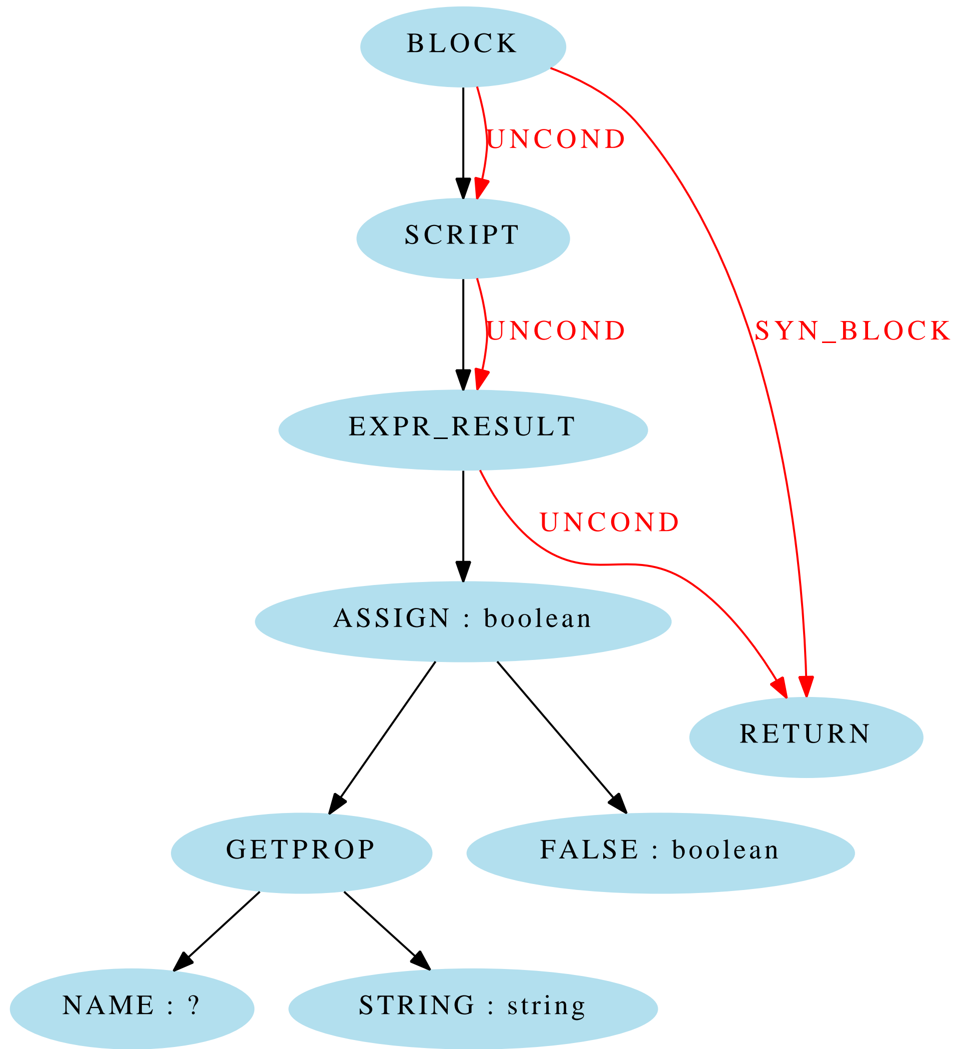


Figure 6: x['y'] = false;

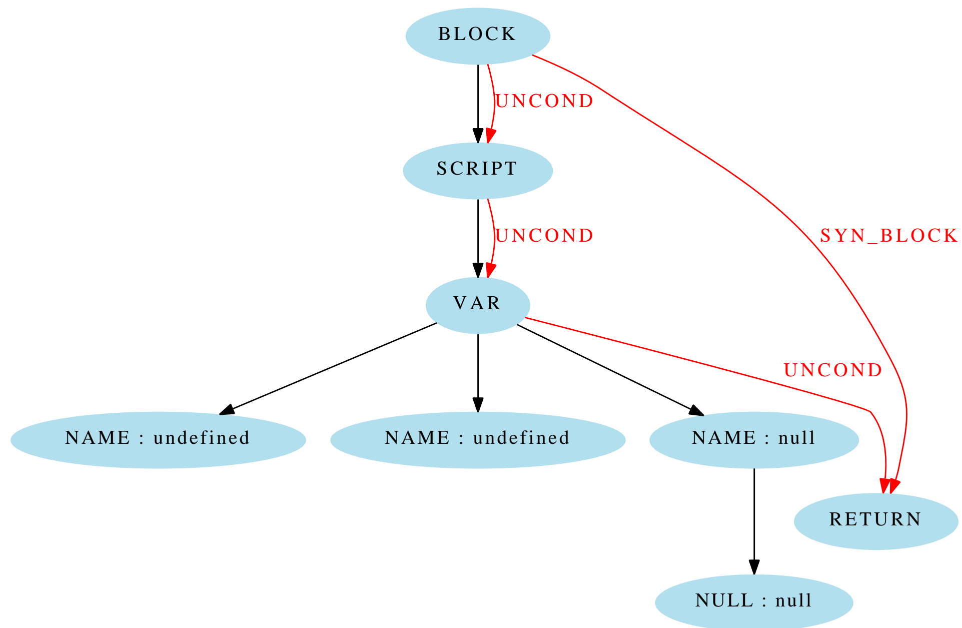


Figure 7: var x, y, z = null;

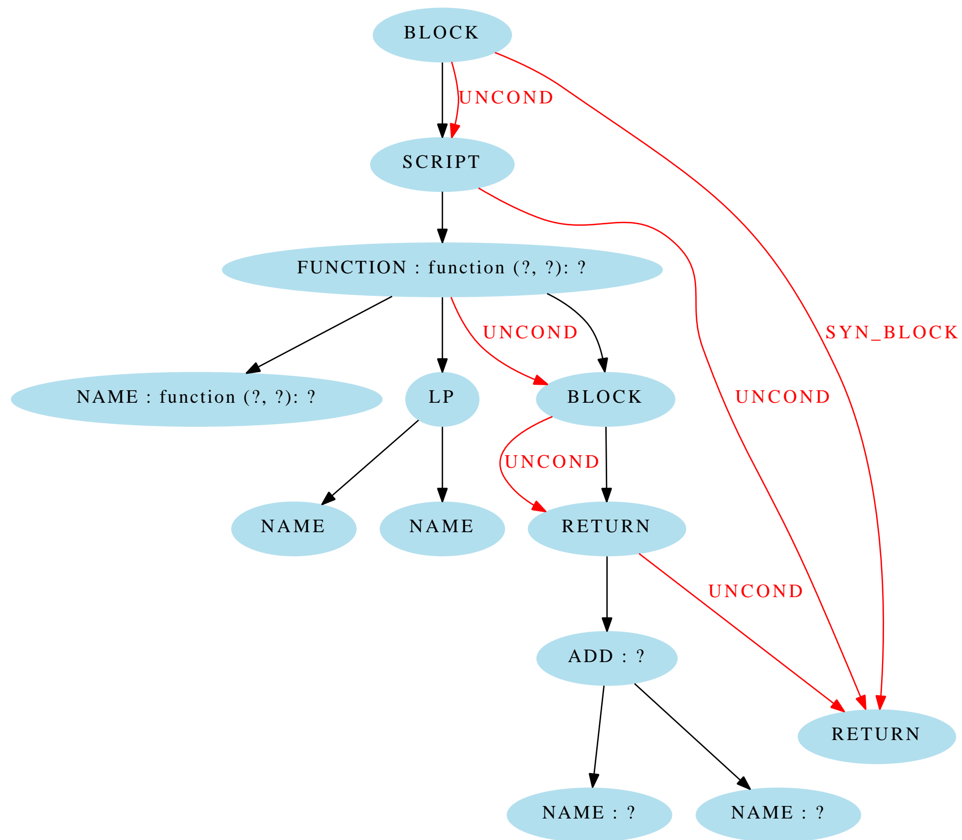


Figure 8: function f(x, y) { return x + y; }

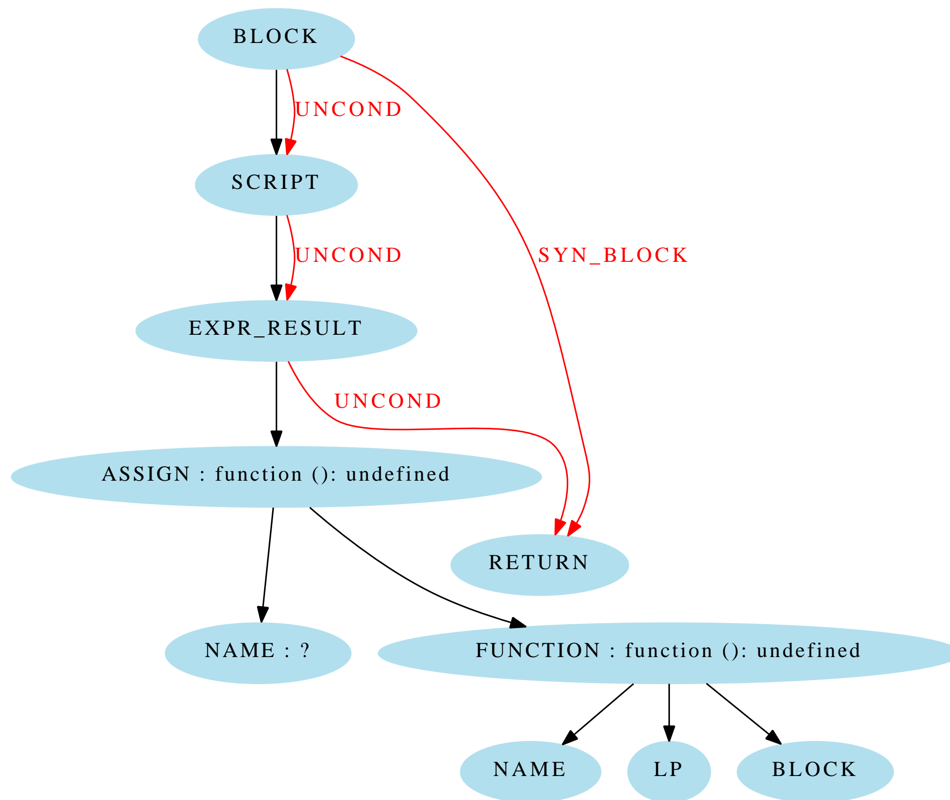


Figure 9: z = function(){};

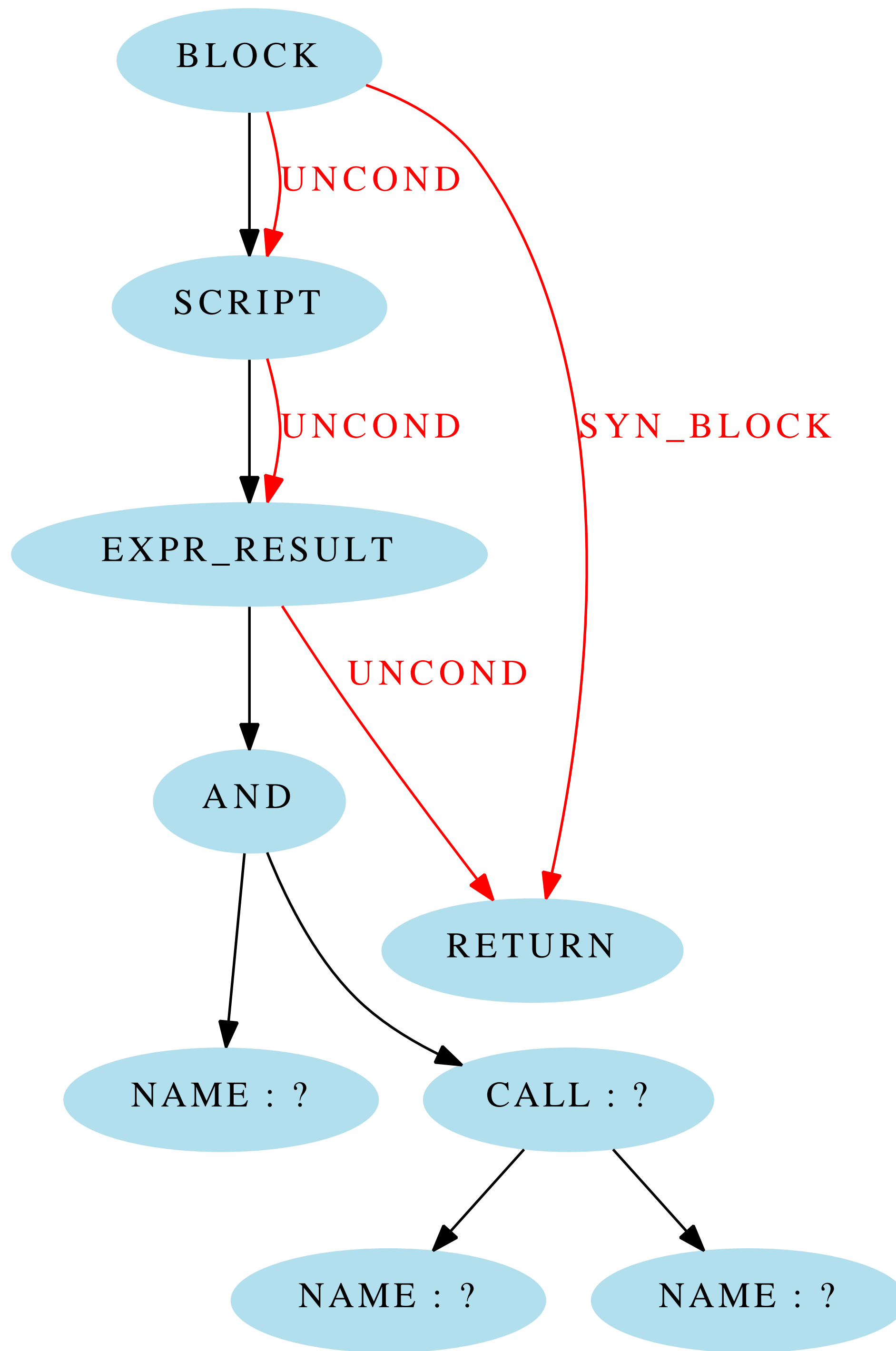


Figure 10: if (x) { y(z); }

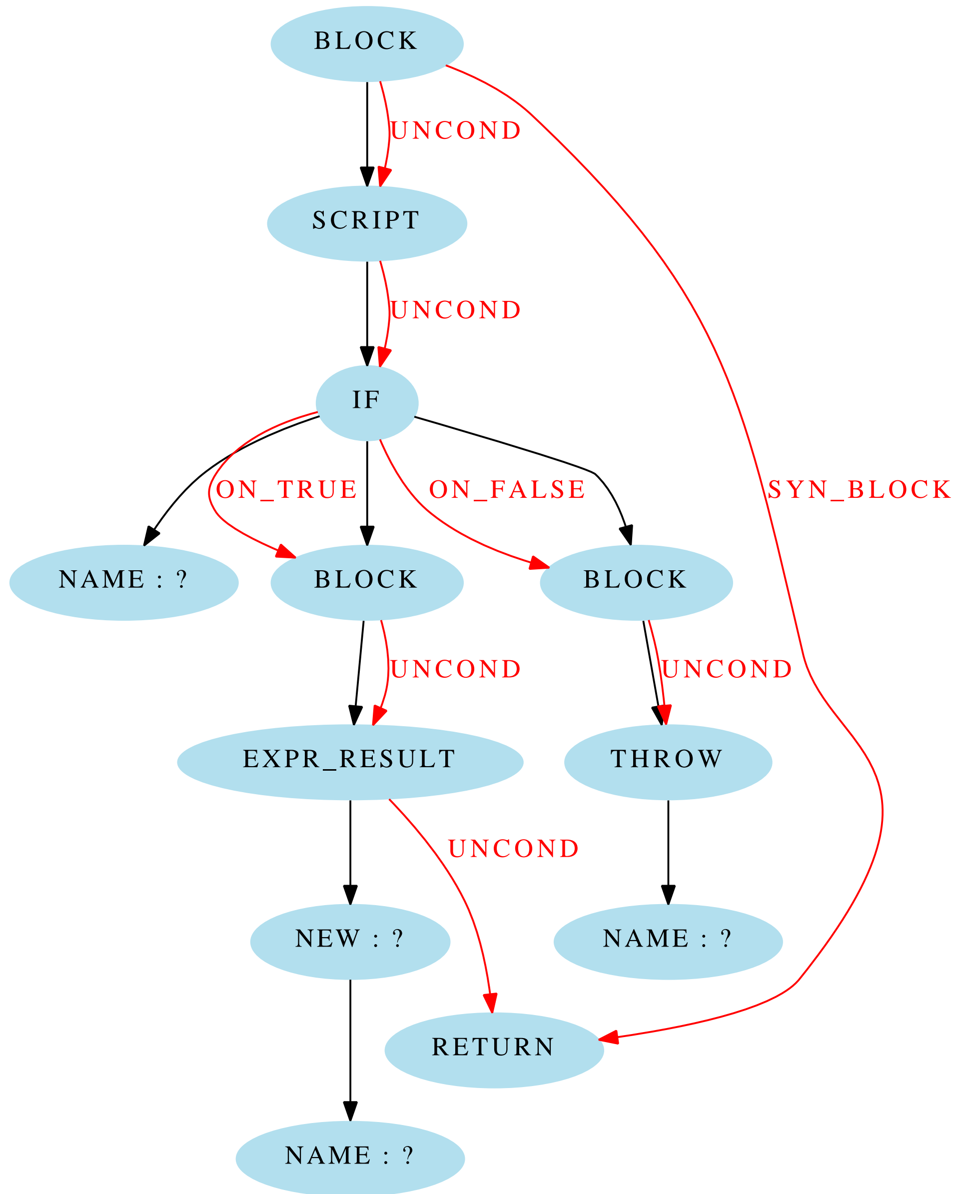


Figure 11: if (x) { (new y); } else { throw z; }

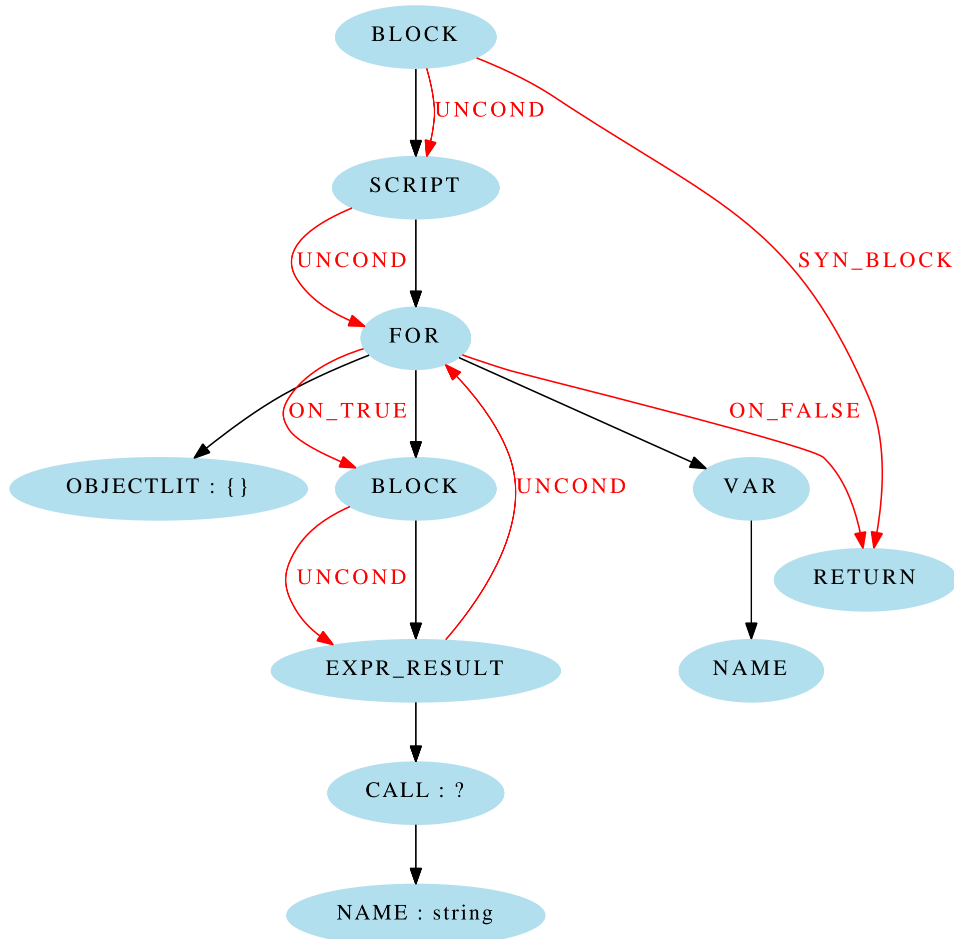


Figure 12: for (var i in {}) { i(); }

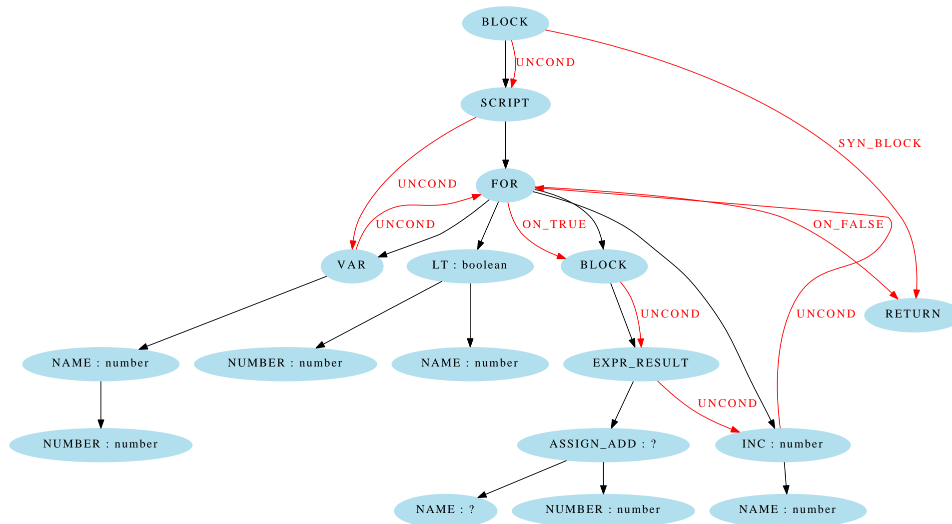


Figure 13: for (var i = 0; i ¡ 10; i++) { x += 1; }

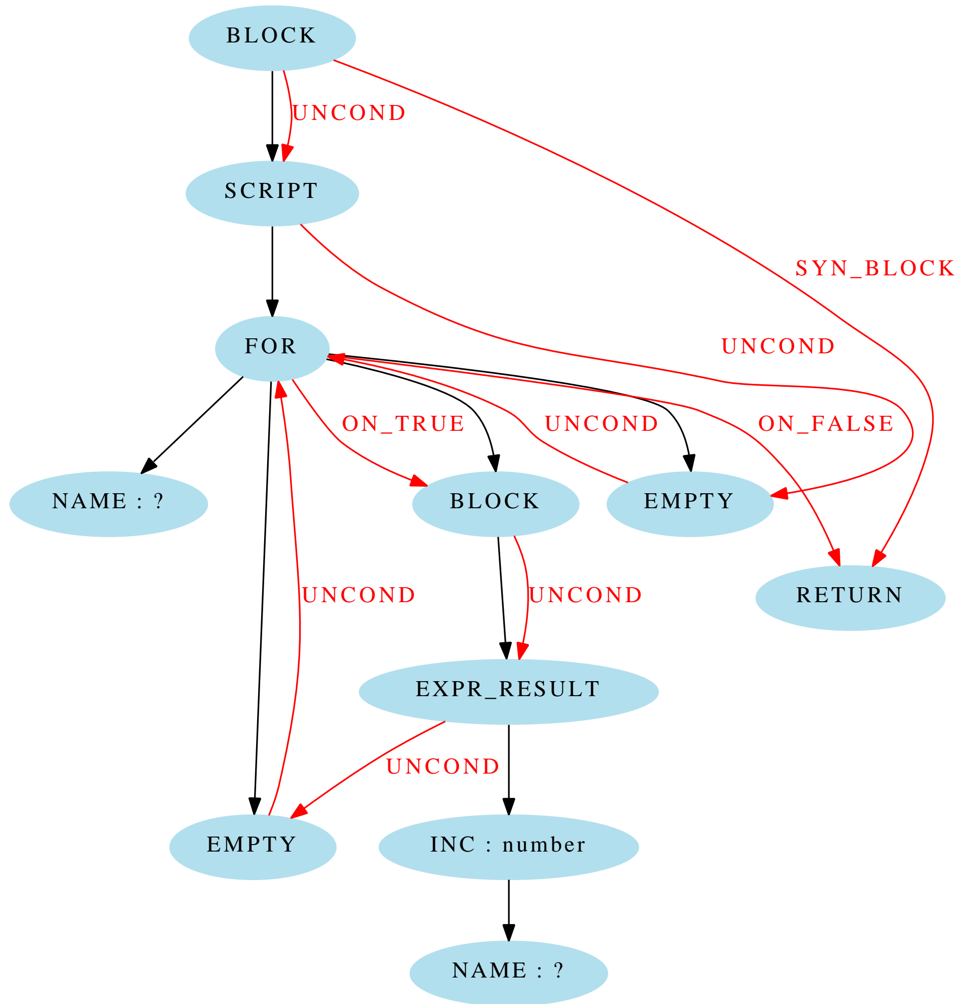


Figure 14: while (x) { x++; }

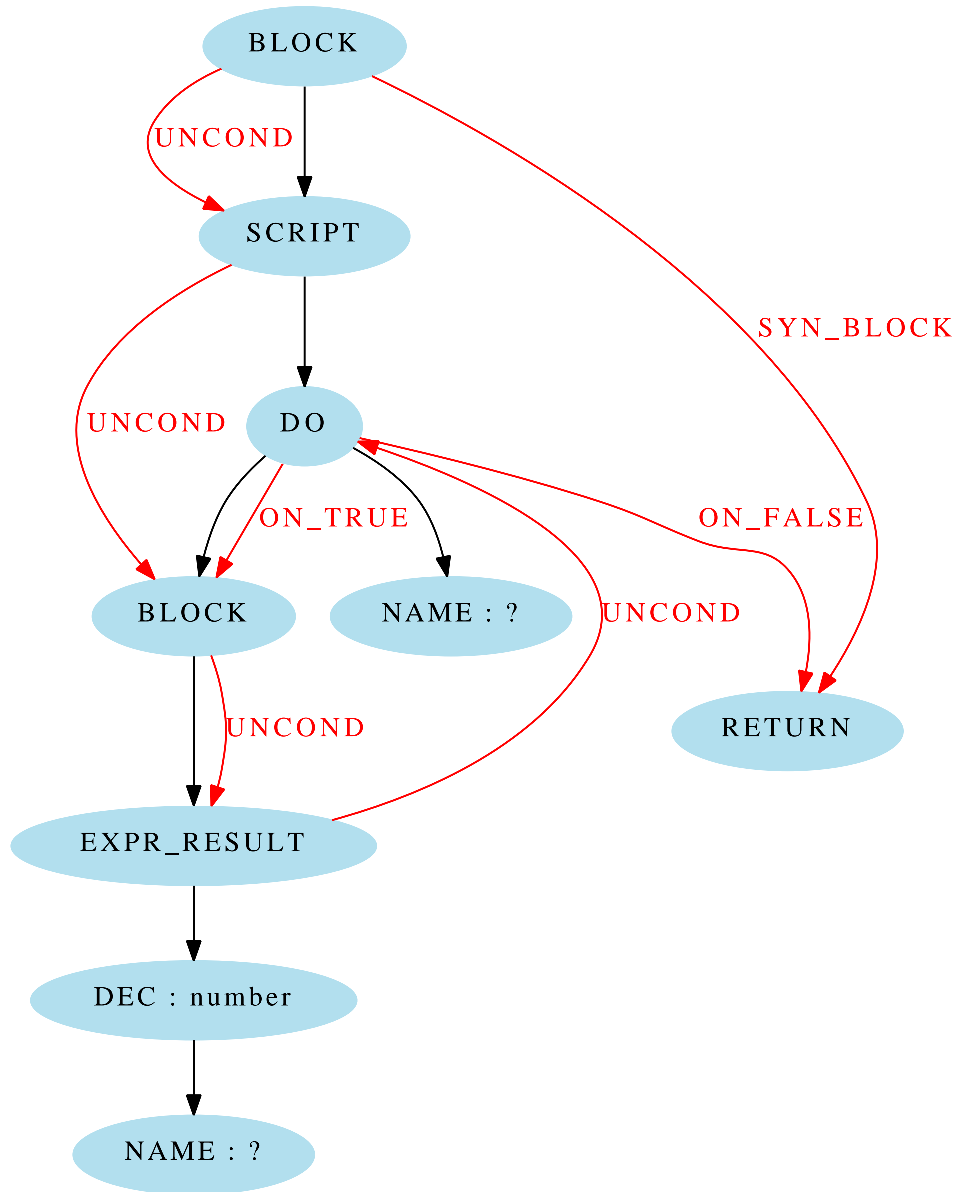
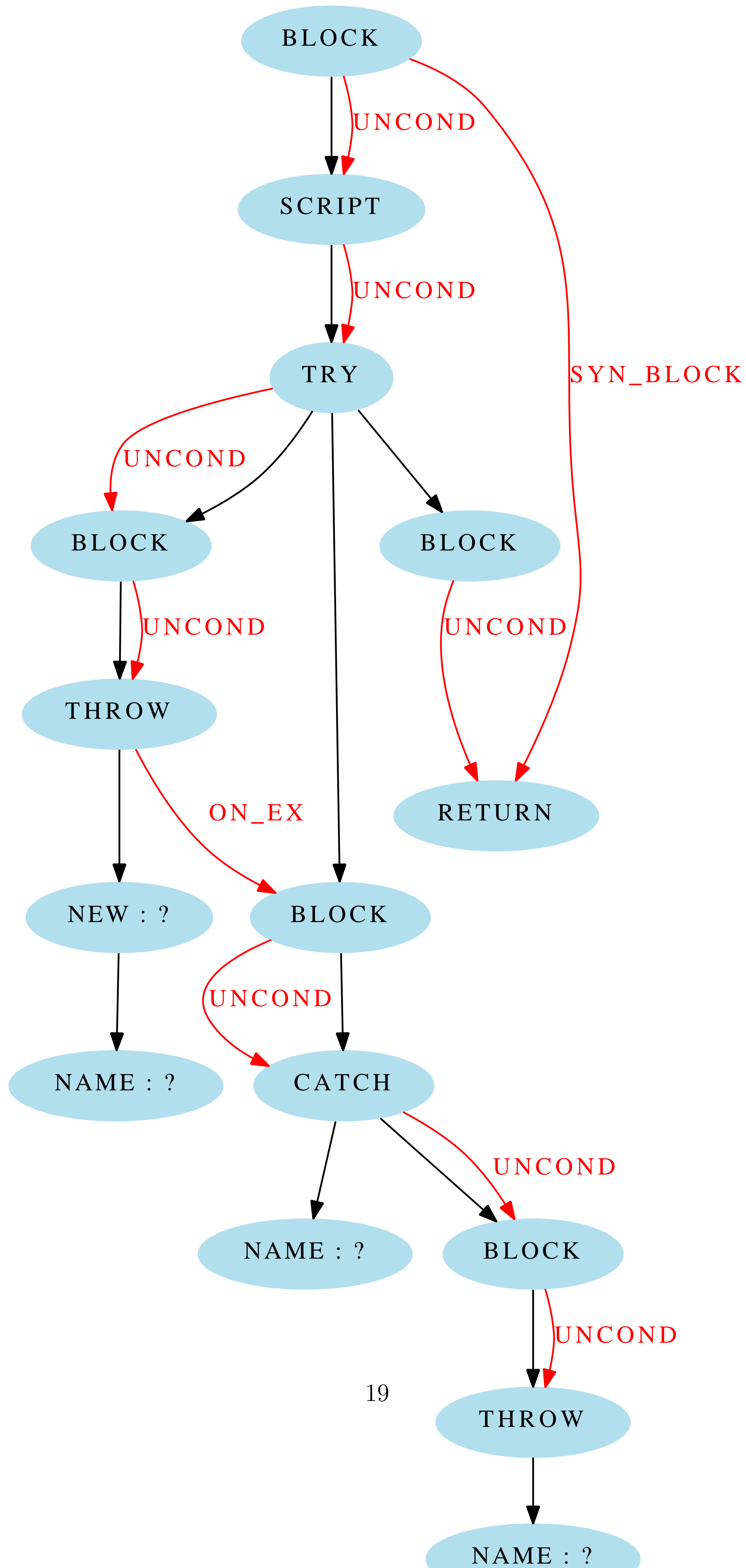


Figure 15: do { –x; } while (x);


BLOCK
UNCOND
SCRIPT
UNCOND
TRY
SYN_BLOCK
UNCOND
BLOCK
BLOCK
UNCOND
UNCOND
THROW
ON_EX
RETURN
NEW : ?
BLOCK
UNCOND
NAME : ?
CATCH
UNCOND
NAME : ?
BLOCK
UNCOND
THROW
NAME : ?

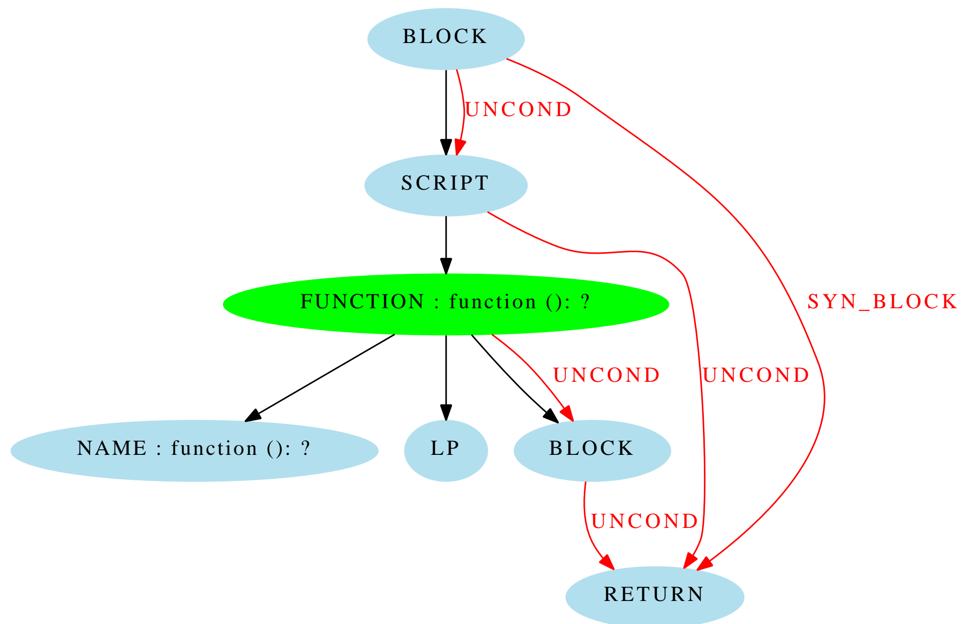


Figure 17: /** This is jsdoc */ function g() { return undefined; }

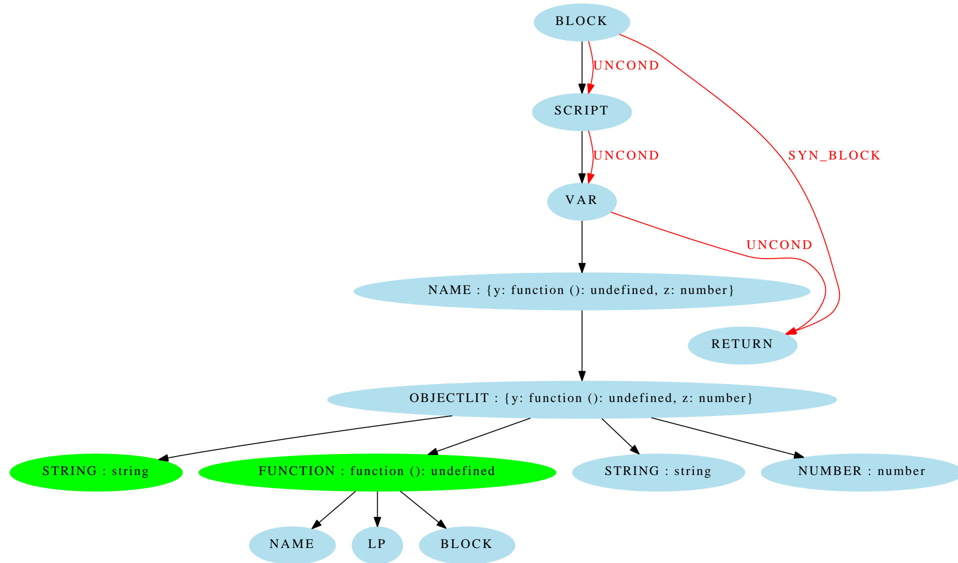


Figure 18: var x = {/** jsdoc */ y: function() {}, 'z': 3};

Figure 19: label: for (i in x) { break label; } i = 5;