TDT4205 Compiler Construction

Recitation Lecture PS3

haavakro@stud.ntnu.no

Department of Computer Science

2023-02-17

Table of Contents

Table of Contents

LR(0) parsers

In the world of LL, LL(0) makes no sense.

LR is different. LR(0) can at least parse *some* languages.

LR consumes input from left to right, but constructs bottom up.

A combination of a stack, and a state machine.

The state machine decides when to **shift** more input onto the stack, or **reduce** the top of the stack, corresponding to some production in the grammar.

SLR

LR(0) gives us no tools to handle **shift/reduce conflicts**.

When the top of the stack is corresponding to a production body, while there are other potential productions that need some more input tokens.

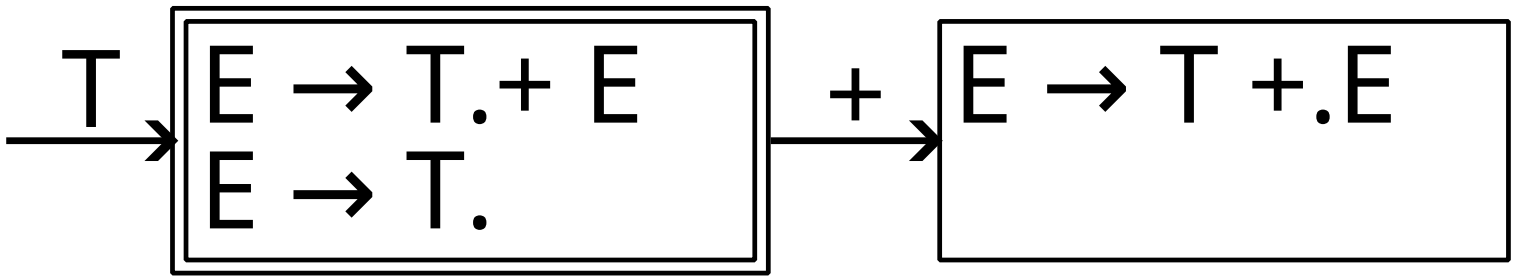


Figure: This is a shift/reduce conflict, since the accepting state has an outgoing edge.

SLR takes a peek at the next input token, to resolve the conflict.

This only works if FOLLOW(E) doesn't contain +.

Table of Contents

Tree Simplification

We will now extend the parser from PS2, with tree simplification!

You can use your own PS2, or download a fresh `ps3_skeleton.tar.gz`.

The skeleton includes some added sample files, and **graphviz** output.

Graphviz!

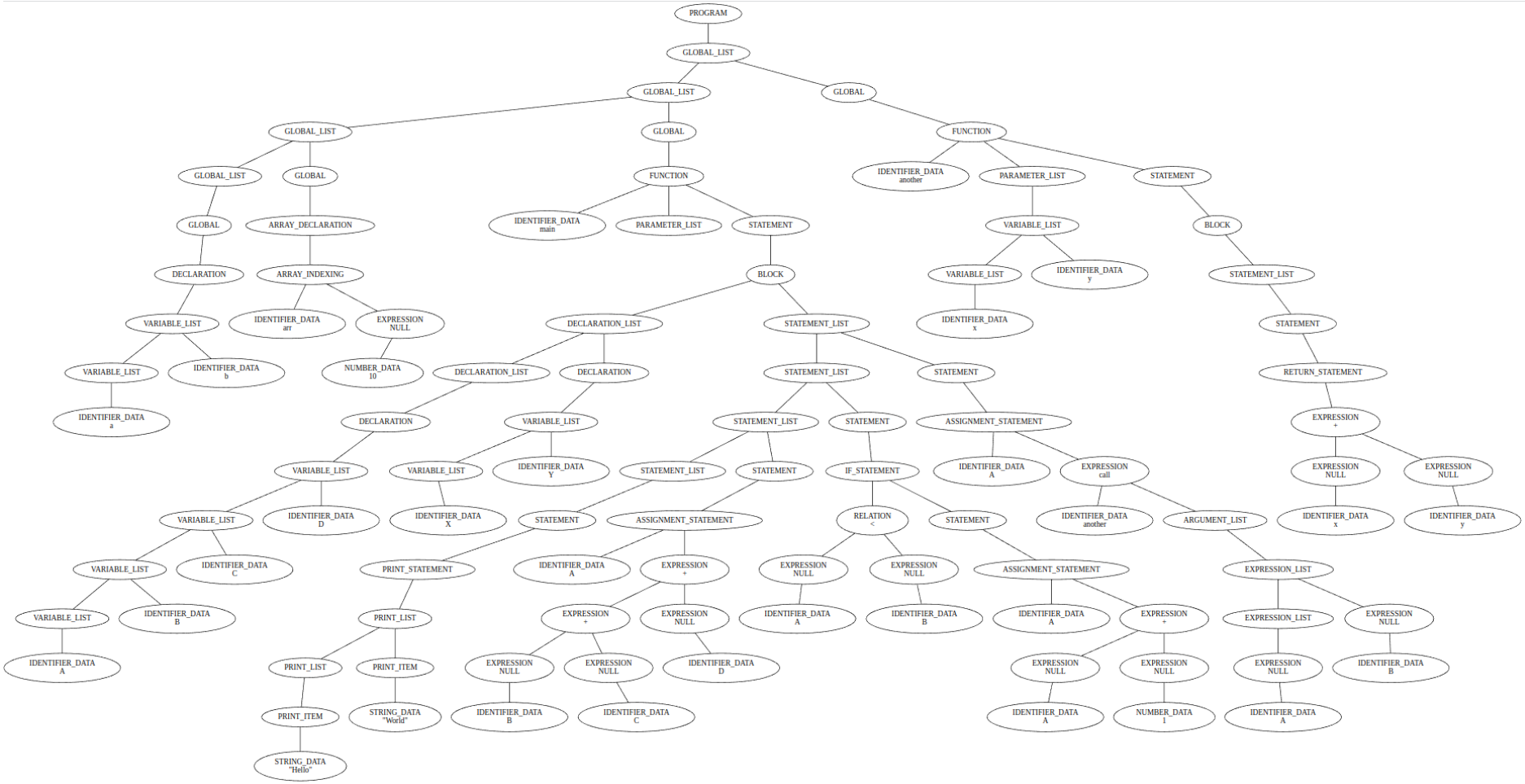


Figure: The file tree_flattening.vsl, before simplification

Graphviz!

Figure: The file tree_flattening.vsl, after simplification

Tip: Incremental implementation

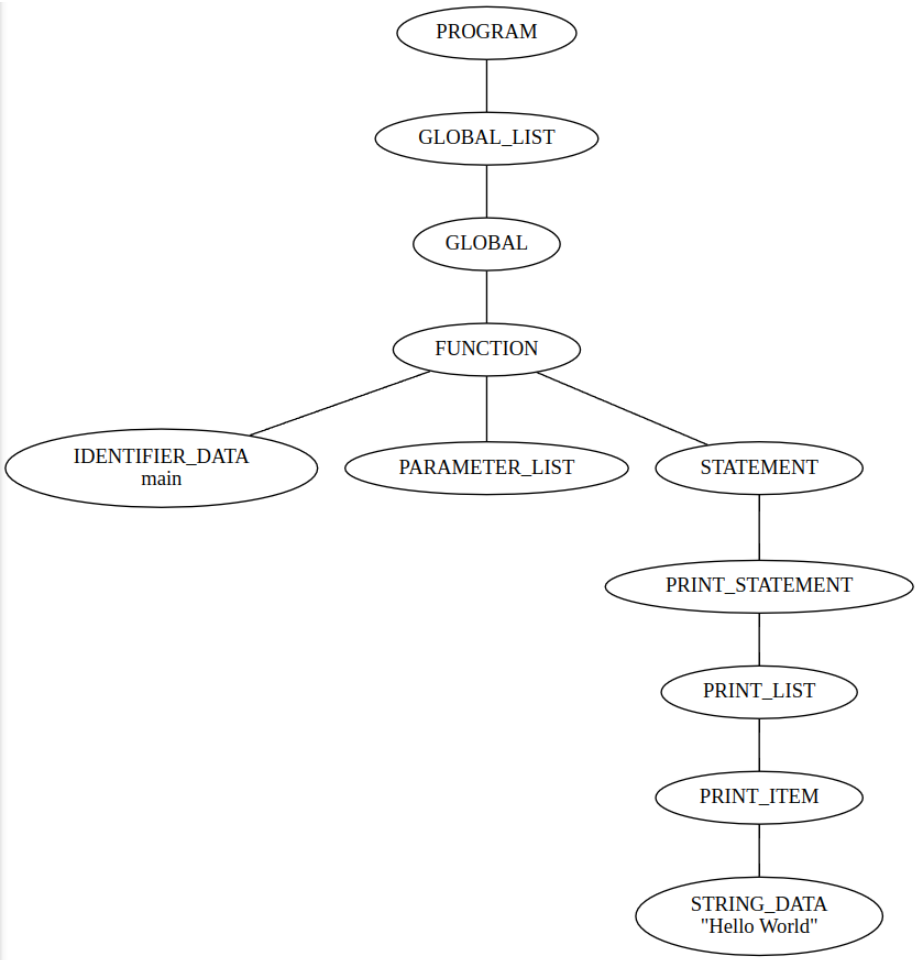
Last exercise, the code would not really compile until you finished the parser.

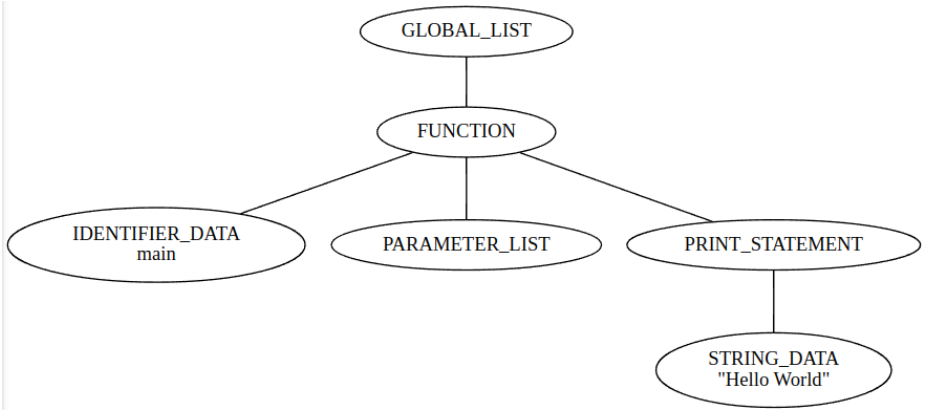
This time it compiles **from the get-go**.

```
tar -xzf ps3_skeleton.tar.gz
cd ps3_skeleton
make
cd vsl_programs
make ps3-graphviz
```

This should make it much easier to test out every step of your code.

Removing unecessary nodes

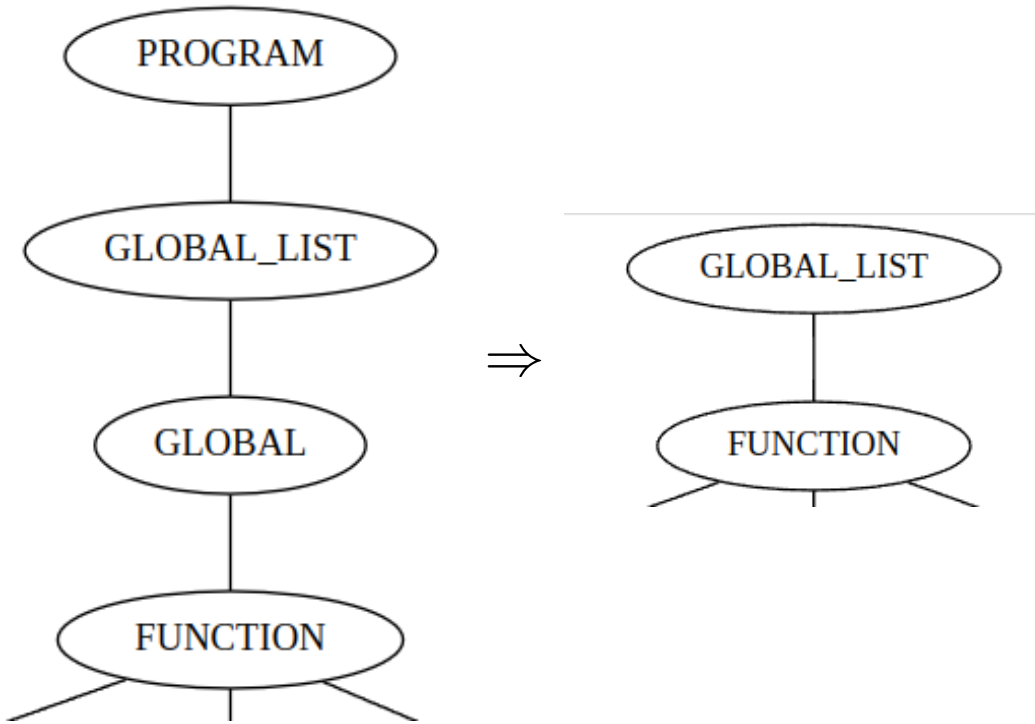




Unecessary nodes

A node with only one child, and no semantic value, can often just be replaced by its child.

This can be seen in the `PROGRAM` and `GLOBAL` nodes.



NB: Remember to finalize the nodes you remove.

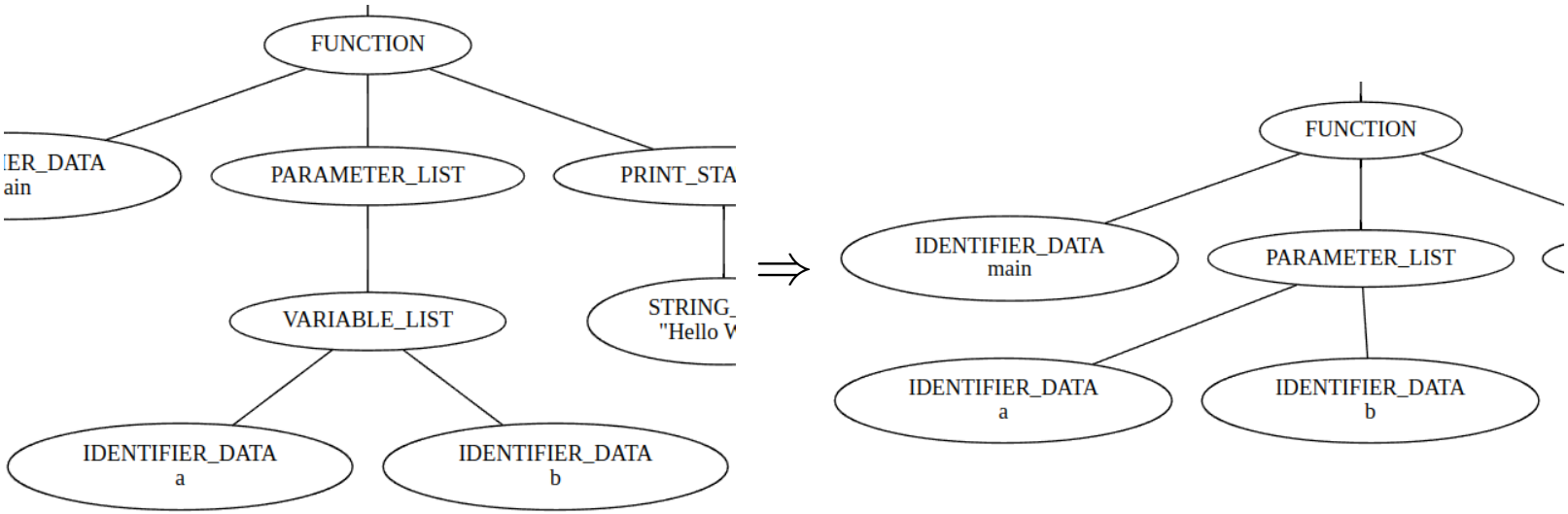
Another type of unecessary node

Again we have a parent with a single child.

The `PARAMETER_LIST` serves as a wrapper, to allow the list to be empty.

We could remove the parent, but the parent has a **better name**.

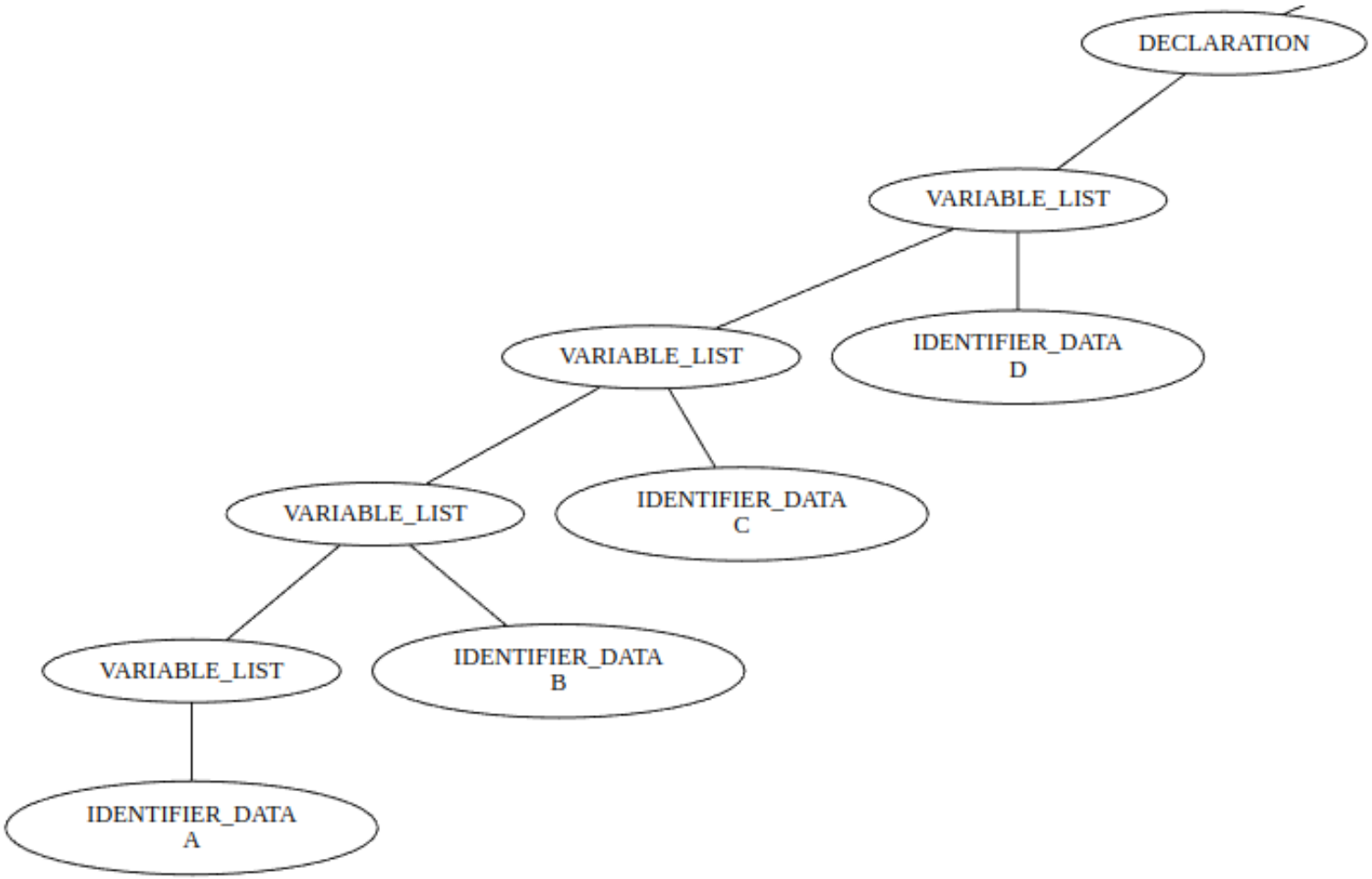
We instead let the `PARAMETER_LIST` "squash" its child, and steal its children.



Flattening lists

Lists are currently stored as linked lists of nodes.

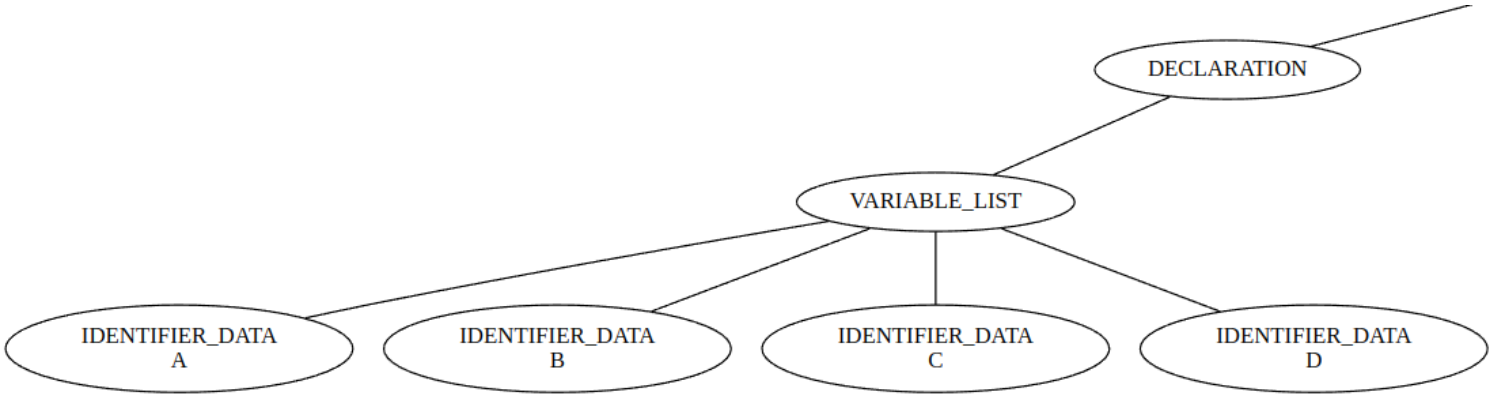
We wish to flatten them into one list node, with all the children.



Flattening lists

There are several ways to do this, but

- Remember to free up the nodes you remove
- Use `realloc` to resize allocations

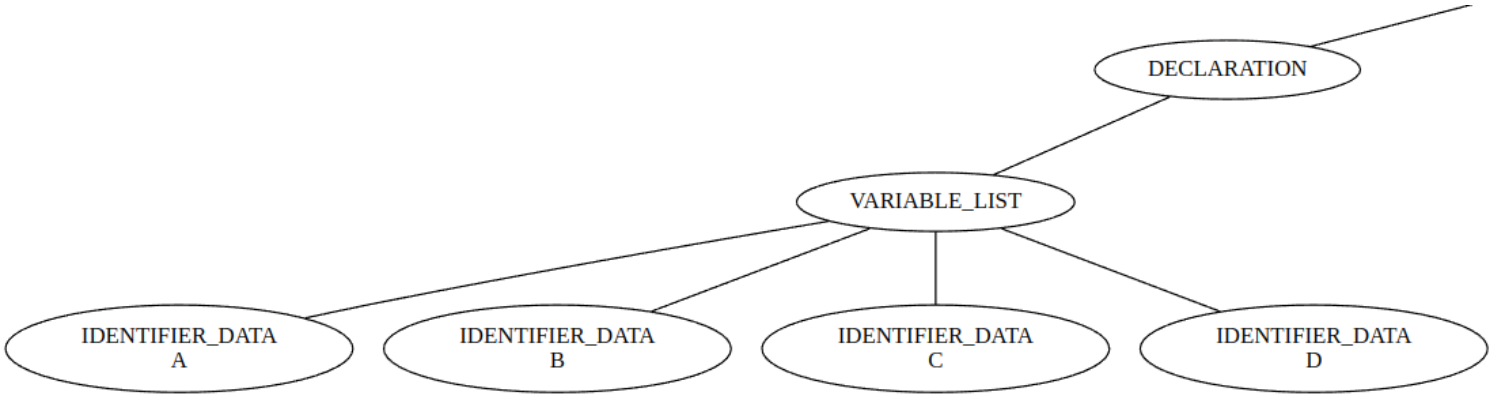


Unecessary node alert!

That `DECLARATION` node has only one child.

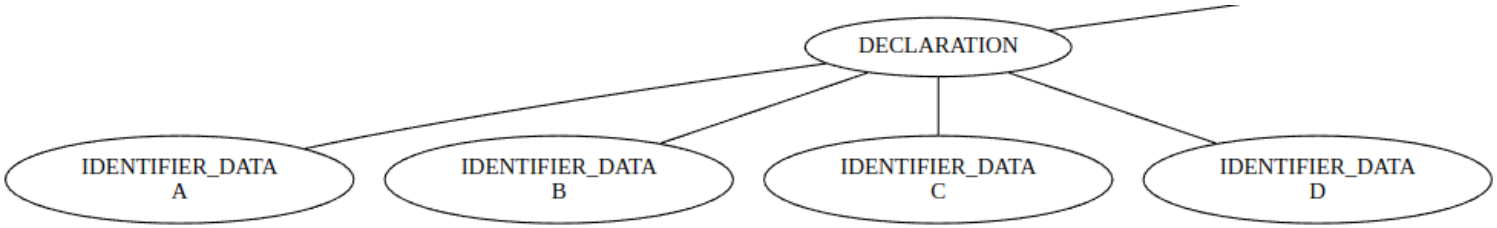
We can now pick between

- Replacing `DECLARATION` with its child
- Let `DECLARATION` squash its child



Squashing after list flattening

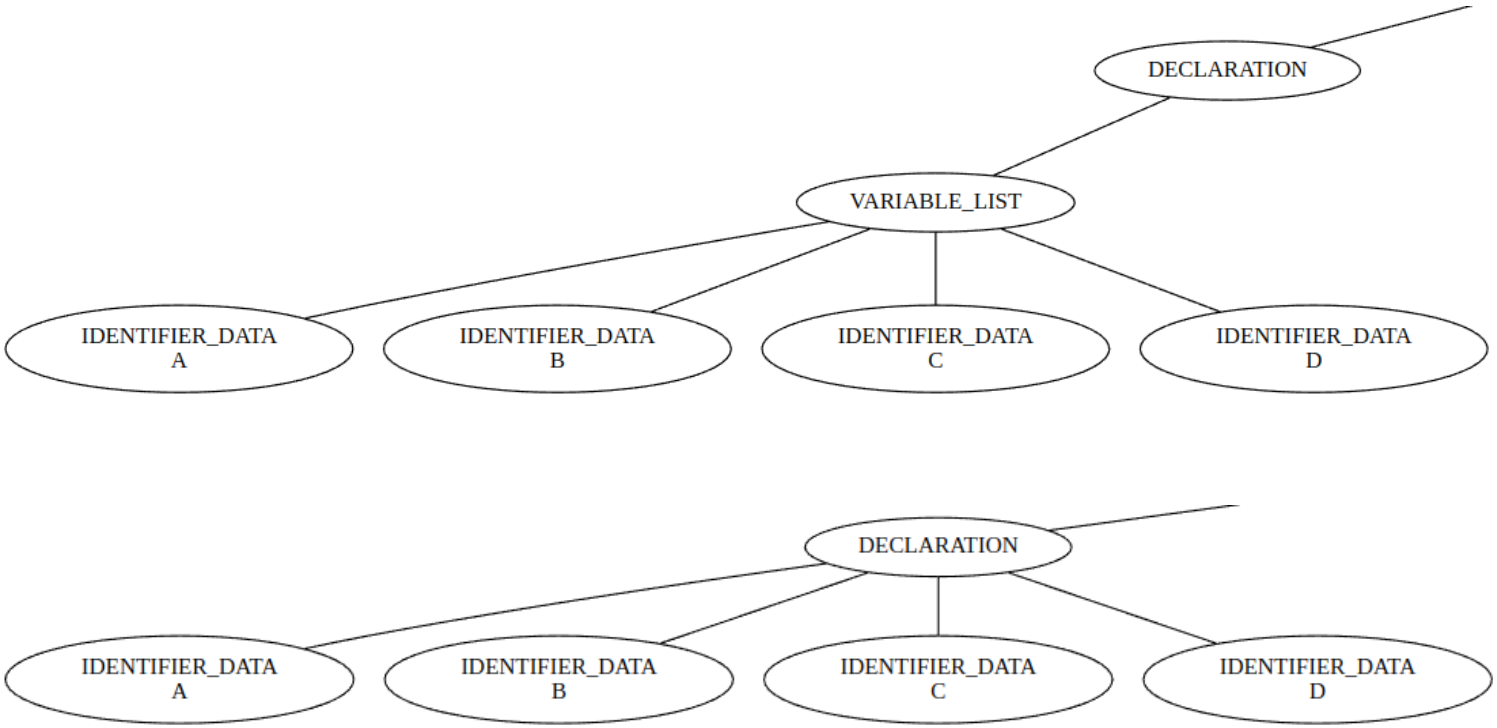
The `DECLARATION` node disposes of its child, and takes all its children as its own.



Order of operations

The simplification function is recursive, and should work bottom up.

Before the `DECLARATION` node squashes, everything below it should already be simplified.



Constant folding

Expression nodes with no operation, can be replaced by their child.

Expression nodes with operations, can only be replaced if all children are `NUMBER_DATA`.

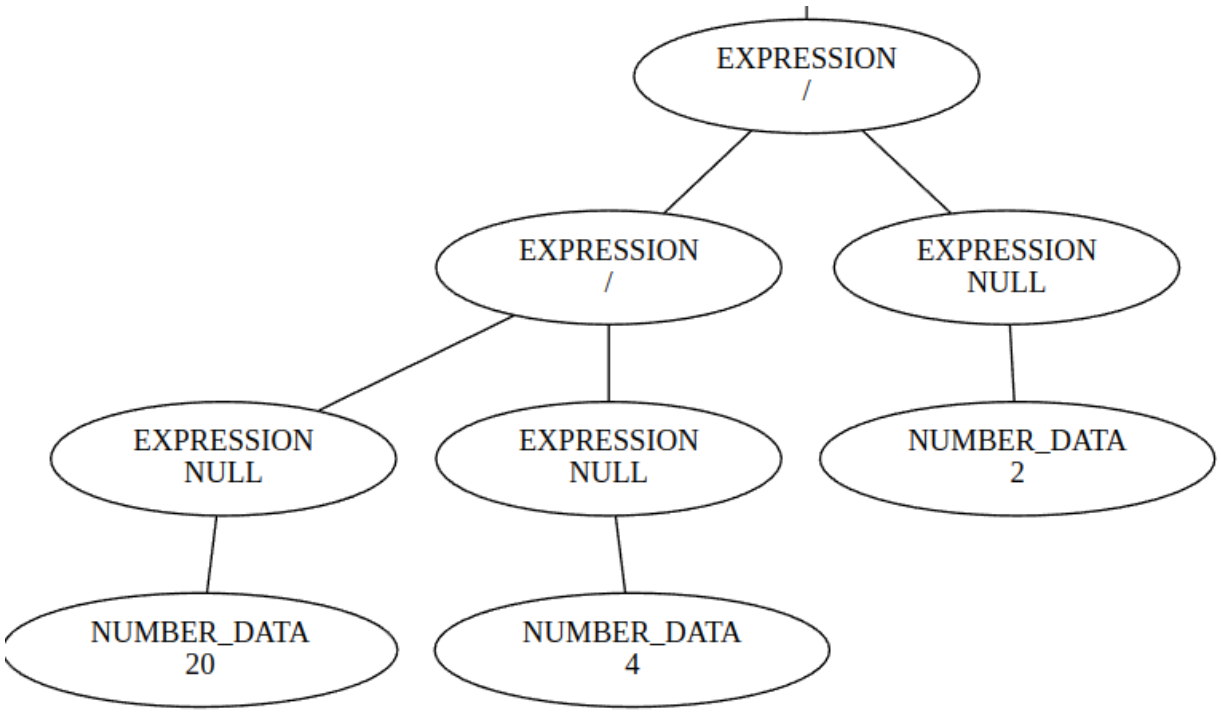


Figure: This should all end up as a single NUMBER_DATA node, with the data 2

Constant folding tips

Check if `data->data` is `NULL` first. `NULL` behaves poorly with `strcmp`.

Use `strcmp(node->data, "+")` etc. to check operators.

Remember to `malloc(sizeof(int64_t))` to let the resulting `NUMBER_DATA` node own its data.

Remember to finalize up all the `EXPRESSION` and `NODE_DATA` nodes you remove.

For loop conversion

We don't want to have two types of loops in the compiler. We convert:

```
for i in 5..N+1
    print "i:", i
```

```
begin
  var i, __FOR_END__
  i := 5
  __FOR_END__ := N+1
  while i < __FOR_END__ begin
    print "i:", i
    i := i + 1
  end
end
```