TDT4205 Problem Set 3

Feb 17, 2023

Answers are to be submitted via Blackboard by **March 2nd**. Please submit your answer as an archive `username.tar.gz` containing a PDF file with answers to theoretical questions, and a code directory like the provided one, containing all files required to build your solution.

Use any resources you can, the recitation lecture slides, asking each other, and Piazza. Start early, and when the deadline approaches, **submit what you have got**, no matter how finished. Remember to cite your sources and collaboration partners wherever they have influenced your work.

1 Bottom/up parsing tables

Consider the following grammar:
$E \rightarrow E + T | T$
$T \rightarrow T * F | F$
$F \rightarrow x$

1.1

Augment the grammar, and construct its LR(0) automaton.

1.2

Identify any states with shift/reduce conflicts in your automaton.

1.3

Is the grammar SLR parseable?
Justify your answer by constructing the SLR parsing table.

2 Implementation — Tree simplification

We will now continue working on the compiler we started in PS2, by simplifying the parse tree created by the parser, into an abstract syntax tree. The resulting tree should be easier to work with in later exercises.

You can use your own solution to PS2 as a starting point, or download a fresh version of `ps3_skeleton.tar.gz`. The skeleton includes some changes, such as stubs for functions, and an optional output mode for printing the tree as a graphviz graph. Graphviz can be installed with `sudo apt install graphviz` on Ubuntu.

Unlike last exercise, the skeleton will compile as is, so it should be much easier to make incremental changes, and test them one by one. Try for instance

```
cd ps3_skeleton
make
cd vsl_programs
make ps3-graphviz
```

and have a look inside `vsl_programs/ps3-simplify`. The `.svg` files show what the tree looks like, for each of the sample programs. Compare these files to those found in the `suggested/` folder, and see the difference.

All the code for this execrise can be written in `tree.c`. The simplification function, `simplify_tree ( node_t *node )`, is a recursive function that takes in a subtree, and returns the root of a simplifed version of the same subtree.

2.1 Eliminate nodes of purely syntactic value

Some node types will only ever have 1 child, and hold no meaningful data. In these cases, we want to either delete the parent, or "squash" the child.

Nodes like `PROGRAM` or `PRINT_ITEM`, can be replaced by their only child, and no information will be lost.

Other nodes, such as `PRINT_STATEMENT`, `DECLARATION` and `PARAMETER_LIST`, have names that clearly communicate what they represent. Replacing them with their only child, can lead to confusing node types in the simplified tree.

To keep the node type that best explains the semantics of a node, we can elect to keep the parent node, "squashing" the child node, and stealing all its children.

See figure 1 for an example of this.

2.2 Flatten list structures

Lists are currently represented as linked lists of nodes, where each node has up to two children. Delete the internal nodes of these list structures, leaving only a single list node, and all the entries as its children.

See figure 1 for an example of a list before and after flattening.

Tip: The size of the `node->children` list will have to change to fit arbitrarily many children. The `realloc()` function lets you resize an existing allocation, or will copy the data into a new allocation for you.

Another tip: Some nodes called `_LIST`, such as `PARAMETER_LIST`, are not actually lists. They are instead optional wrappers around some other list node. In these cases, the child is a good candidate for "squashing", as described in task 2.1.

2.3 Constant folding

Expression nodes that don't perform any operation, and only have one child, can be replaced by their child, no matter what.

If the expression performs some operation, then it can only be replaced if the values of all its children are known at compile time. The result of the operation can be calculated, and the expression node be replaced by a `NUMBER_DATA` node holding the result.

2.4 Replacing for-loops

The compiler supports two types of loops. `while`-loops keep going until a comparison turns false, and `for`-loops count from a start value until (but not including) an end value is reached.

To simplify the backend of the compiler, we only support `while`-loops. We must therefore convert `for`-loops as part of the tree simplification.

See `node_t* replace_for_statement ( node_t* for_node )` for examples of how a for-loop can be converted to a while-loop in source. Sadly we are now working with nodes, and have to do the same transformation by manually creating nodes for everything.

This final part is quite tricky, so try to help each other out. It is not paramount that you finish this part of the problem set, but make an honest attempt. You can use the graphviz output to compare your `for.svg` to `suggested/for.svg` while working.

Valgrind and memory correctness

Modiying tree structures by hand in C is tricky business. Here are some things to watch out for:

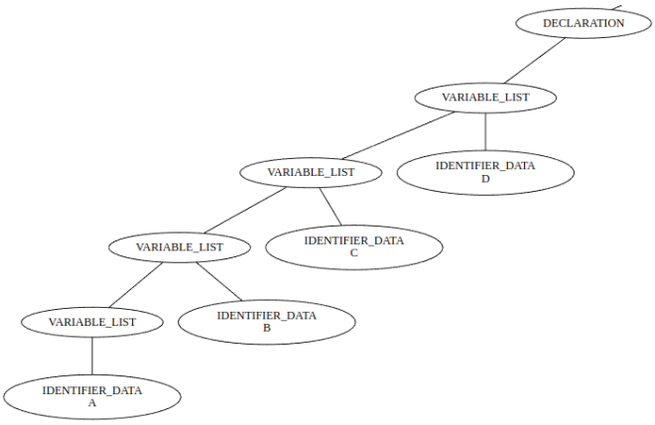
- Use-after-free:
  - Freeing a node you still use in your tree.
  - Freeing the `node->children` or `node->data` of a node in the tree.
- Double free during cleanup:
  - Using the same node twice in the tree.
  - Re-using the same `node->children` or `node->data` in multiple nodes.
- Memory leaks:
  - Forgetting to free a node you no longer use in the tree.
  - Replacing the `node->children` list, or `node->data` without freeing the old value.

We won't be checking your code for these, but you risk getting some weird behaviour if you have use-after-free bugs.

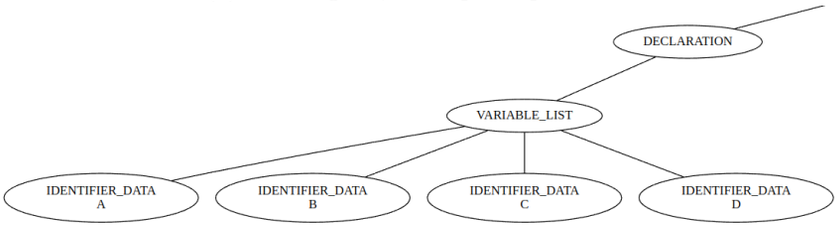
To detect all of these, run using `valgrind`. Try for instance

```
valgrind src/vslc < vsl_programs/ps3-parser/for.vsl
```

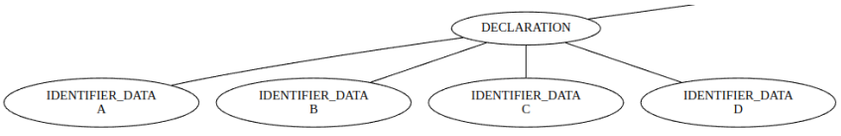
and it will tell you about memory bugs, and memory leaks.



(a) The original, unsimplifed parse tree



(b) The parse tree after list flattening

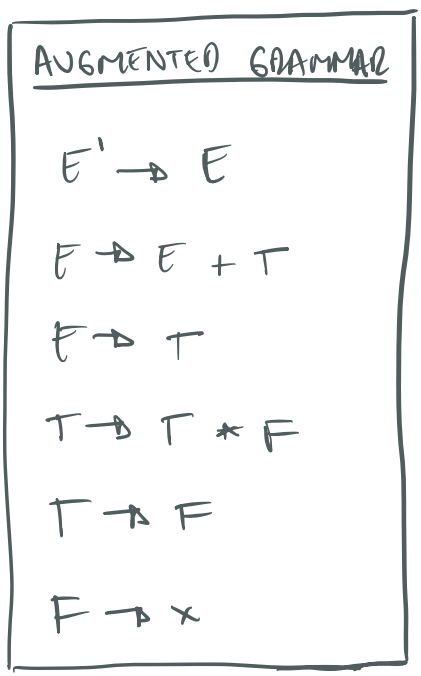


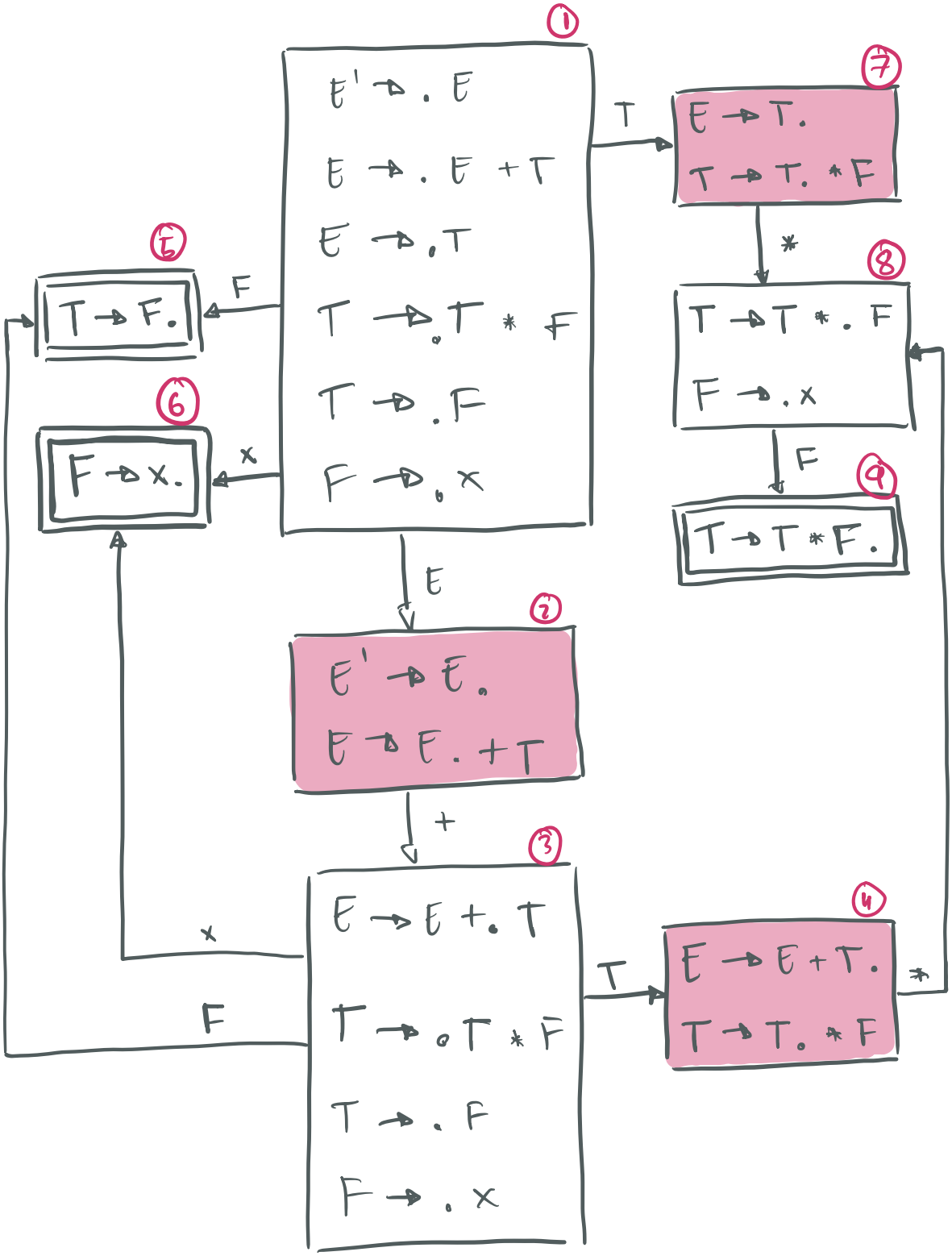
(c) The tree after the DECLARATION node has squashed its only child, and stolen its children

Figure 1: Examples steps of simplifying a subset of tree_flattening.vsl

1.1

Augment the grammar, and construct its LR(0) automaton.




1
E' → . E
E → . E + T
E → . T
T → . T * F
T → . F
F → . x
T
7
E → T .
T → T . * F
*
8
T → T * . F
F → . x
F
9
T → T * F .
F
5
T → F .
6
F → x .
x
E
2
E' → E .
E → E . + T
+
3
E → E + . T
T → . T * F
T → . F
F → . x
T
4
E → E + T .
T → T . * F
*
x
F

1.2

Identify any states with shift/reduce conflicts in your automaton.

HIGHLIGHTED IN RED IN 1.1.

1.3

Is the grammar SLR parseable?
Justify your answer by constructing the SLR parsing table.

IT IS SLR PARSEABLE.

FOLLOW

E	$, +
T	$, *, +
F	$, *, +

A) $E' \to E$ → ACCEPTING

1) $E \to E + T$
2) $E \to T$
3) $T \to T * F$
4) $T \to F$
5) $F \to x$

	*	+	x	$	E	T	F
1			s6		g2	g7	g5
2		s3		A			
3			s6			g4	g5
4	s8	r1		r1			
5	r4	r4		r4			
6	r5	r5		r5			
7	s8	r2	s6	r2			
8							g9
9	r3	r3		r3			