POP SS19 Homework 2 Documentation

Robert Ernstbrunner, 01403753

May 3, 2019

Since I cannot attend the next lecture to present my work I tried my best to explain in great detail what was done here.

Introduction

In this assignment the abstract syntax tree (*AST*) for executable statements of *C* functions had to be build. To verify the correctness of the *AST* the output was visualized with help of the *dot* language and the *graphviz* environment. This document describes the structure of the *AST* and gives a general overview of the implementation.

AST structure

The structure of the abstract syntax tree as well as the interface for generating the *AST* and *dot* file are defined in the `intermediate.h` header file (Figure 1).

struct ast

The struct of the tree can be observed in lines 4 - 10 in Figure 1. The naming conventions are similar to those in the *Flex and Bison* guide [1].
I decided to build a binary (instead of a ternary) tree. Each node comprises a *nodetype* of type `int` that indicates whether the node is a decision node (*nodetype* = 'D' for decision node)[1], a leaf node of type `double` (*nodetype* = 'C' for constant) or a leaf node of type `char *` (*nodetype* = 'I' for identifier).
Decision nodes store their label in the `char *` variable and either the left or right or both pointers to subtrees are set. Leaf nodes either assign the `char *` or the `double` variable and the pointers to subtrees are set to `NULL`.

[1]Notice the implicit type conversion from `char` to `int`.

interface description

- The *newast(. . .)* method creates and returns a decision node. Thus, it needs a label and a left and right subtree as input (where one of them can be set to `NULL`).

- The *newnum(. . .)* method creates and returns a leaf with a `double` value. This method is called whenever a `CONSTANT` is returned from the lexer.

- The *newid(. . .)* method creates and returns a leaf with a `char *` value. This method is called whenever it is appropriate, e.g., when an `IDENTIFIER` or keyword is returned from the lexer.

- The *generate_dot(. . .)* method creates the `test_function.dot` file. It then makes one call to *write_tree(. . .)* and thereby starts a chain of recursive calls that fill up the `test_function.dot` file with nodes and edges.

- The *write_tree(. . .)* method labels nodes and the *write_node(. . .)* method links nodes. Both routines call each other recursively until every node has been visited in a depth first search manner.
  As long as the current node is not a leaf node the *write_tree(. . .)* method computes either the number of the left child node with `2 · nodenum`[2] or the number of the right child node with `2 · nodenum + 1`, or, in case of two subtrees, both and then calls the *write_node(. . .)* routine(s) accordingly. After linking the parent node to the current node the *write_node(. . .)* method calls the *write_tree(. . .)* method again. This approach was inspired by the bison calculator by Jim Mahoney[2].

- The *free_tree(. . .)* method frees all the nodes that were allocated during the parsing process. Unfortunately, I couldn't check with *Valgrind* if all memory is freed correctly since I didn't have the time to set up my own environment with *flex* and *bison*.

[2] `nodenum` denotes the current node number and is 1 for the root node. calls the *write_node(. . .)* routine.

```c
#ifndef INTERMEDIATE_H
#define INTERMEDIATE_H

struct ast {
  int nodetype;
  char id[10];
  double d;
  struct ast *l;
  struct ast *r;
};


/* build an AST */
// decision node
struct ast *newast(char *id, struct ast *l, struct ast *r);

// leaf
struct ast *newnum(double d);
struct ast *newid(char *id);

/* generate the dot file for the AST */
void generate_dot(struct ast *e);

/* annotated functions for recursive calls */
void write_tree(struct ast *e, FILE *dotfile, int nodenum);
void write_node(struct ast *e, FILE *dotfile, int nodenum, int parentnum);

/* delete and free the AST */
void free_tree(struct ast *);

#endif // INTERMEDIATE_H
```

Figure 1: itermediate header file

Design decisions

Parser.y

The parser.y file is probably the most interesting file in this assignment. It contains three sections (declarations, rules and auxiliary procedures) and is the central component for parsing.

declarations In order to be able to return a complete parse tree of type *struct ast** in the main function of the third section, an additional parameter for the *yyparse(...)* method in line 9 of Figure 2 has to be defined. This also affects the definition of *yyerror(...)* in line 6 where the same parameter must be incorporated. The debug variable in line 8 is later used in the auxiliary procedures section and will therefore be explained later on. The *%union* construct in lines 13 - 17 is used to tell bison what symbols have what types of values.[3] E.g., the `CONSTANT` token in line 19 is assigned the value *d* of type `double` with the < > command. The types (nonterminals) *primary_expression* and *additive_expression* get the value *astree* of type *struct ast** in line 22.

rules In the rules section probably most of the 'dull' work was done. Out of the countless rules I highlight just one typical example in Figure 3. Right besides the derivation in line 4 the dollar-notation in brackets is used to link the nonterminal *statement* (represented by $1) of type *struct ast** to the rest of the tree (represented by $$). On the further right I defined some debug output for when the debug flag in section three is set. In order to link multiple statements in the *AST* the *newast(...)* method is used to create a new `STATEMENT` node labeled `'STMT'`.
The root of the *AST* was set by defining an additional entry point. In lines 8-9 the *external_declaration* symbol derives to the *parse_tree* symbol and the *AST* is set in the brackets to the right. `parse_tree` is then derived to the same symbols as the `eternal_declaration` usually would. This (supposedly) does not affect the functionality of the grammar. However, it should be mentioned that the entry point might be set a bit too high in the parse hierarchy and could be moved further down.

auxiliary procedures In this section the main method is defined. First of all, a parameter for custom debug output was added in lines 10 - 12 in Figure 4. Furthermore, the *AST* struct is declared in line 14 and its address is given to *yyparse*. After successful parsing the *AST* is passed on to the *generate_dot(...)* method in order to produce the *test_function.dot* file. Afterwards all the allocated memory is (supposedly) freed with the *free_tree(...)* method.

[3] the default value is `int`.

```
// declarations
...
%{
#include "intermediate.h"
...
int yyerror(struct ast **astree, char *s);

int debug = 0;
%}

%parse-param {struct ast **astree}

%union {
  struct ast *astree;
  double d;
  char *id;
}

%token <d> CONSTANT ...
%token <id> IDENTIFIER STRING_LITERAL SIZEOF ...
...
%type <astree> primary_expression additive_expression ...
...
%% // rules
```

Figure 2: declarations (parser.y)

```
%% // rules
  ...
statement_list
  : statement { $$ = $1; if(debug)printf("statement_list : statement\n"); }
  | statement_list statement { $$ = newast("STMT", $1, $2); }
  ;
  ...
external_declaration
  : parse_tree { *astree = $1; }
  ;

parse_tree
  : function_definition
  | declaration
  ;
  ...
%% // auxiliary procedures
```

Figure 3: rules (parser.y)

```
%% // auxiliary procedures
...
int main(int argc, char **argv)
{
  ...
        /*****************************/
        /* specify your own option here */
        /*****************************/

         case 'd':
           debug = 1;
           break;
  ...
  struct ast *astree;
  yyresult = yyparse(&astree);
  ...
  generate_dot(astree);
  free_tree(astree);
  ...
}
```

Figure 4: auxiliary procedures (parser.y)

Function evaluation and visualization

The non trivial test function that was used as example for the *AST* is outlined in Figure 5. I incorporated `IF`, `IF ELSE`, `SWITCH`, `FOR`, `WHILE` and `DO WHILE` statements which can be compound and nested as well.

The `makefile` from the first assignment was used and extended with the rule

```
png:
    dot -Tpng test_function.dot -o test_function.png
```

in order to produce the *.png* file from the *.dot* file. Finally, the visualization of the *AST* is presented in Figure 6. The abort condition of the `WHILE` and `FOR` statements is given in the left subtree while the compound statement of the `WHILE` and `FOR` statements is linked as the right subtree. For the `DO WHILE` statement the branches are switched so that the abort condition is represented in the right subtree. This is due to the fact that, in order to read the code in the right order, the *AST* must be traversed in DFS from left to right. Also, while the subtree for the `IF` statement without `ELSE` branch is structurally similar to the `FOR`/ `WHILE` statements, the `IF ELSE` statement probably needs further explanation. The root of this statement is the `IF` node. To the left is the condition while to the right there is another node denoted as `<- IF | ELSE ->` that comprises the `IF` part in the left subtree and the `ELSE` part in the right subtree.

Additional notes

The *scanner* should also be mentioned, since it had to be adapted form the first assignment as well. Initially, the *scanner* only returned tokens, but not the values that belonged to them. Therefore, in some cases, the *yytext* value had to be stored in the specified values of *yylval*, i.e., for `IDENTIFIERES` the content from *yytext* was stored in *yylval.id* and for `CONSTANTS` the value from *yytext* was stored in *yylval.d*.

The makefile should be used by typing

1. `make` (to build the *cparser* executable)
2. `./cparser test_function.c` (to create the *.dot* file)
3. `make png` (to create the *.png* file)
4. `make clean_all` (to clean everything)

```
void func(int i, double y, double x) {

  if (i  <= 12) {
    switch(i){
      case 19:
        break;
      case 20:
        516 + y * 110 / (x - 1);
      default:
        ;
    }
  } else if (i > 0) {
    25.8069 * 25.8069;
  }

  for (i = 0; i < 11; ++i){
    for (j = 11; j > i; j--) {
      do {
        (ab - 240) / 400;
      } while(x < 10);

      while (y != 5)
        y++;
    }
  }

}
```

Figure 5: evaluated function *func*

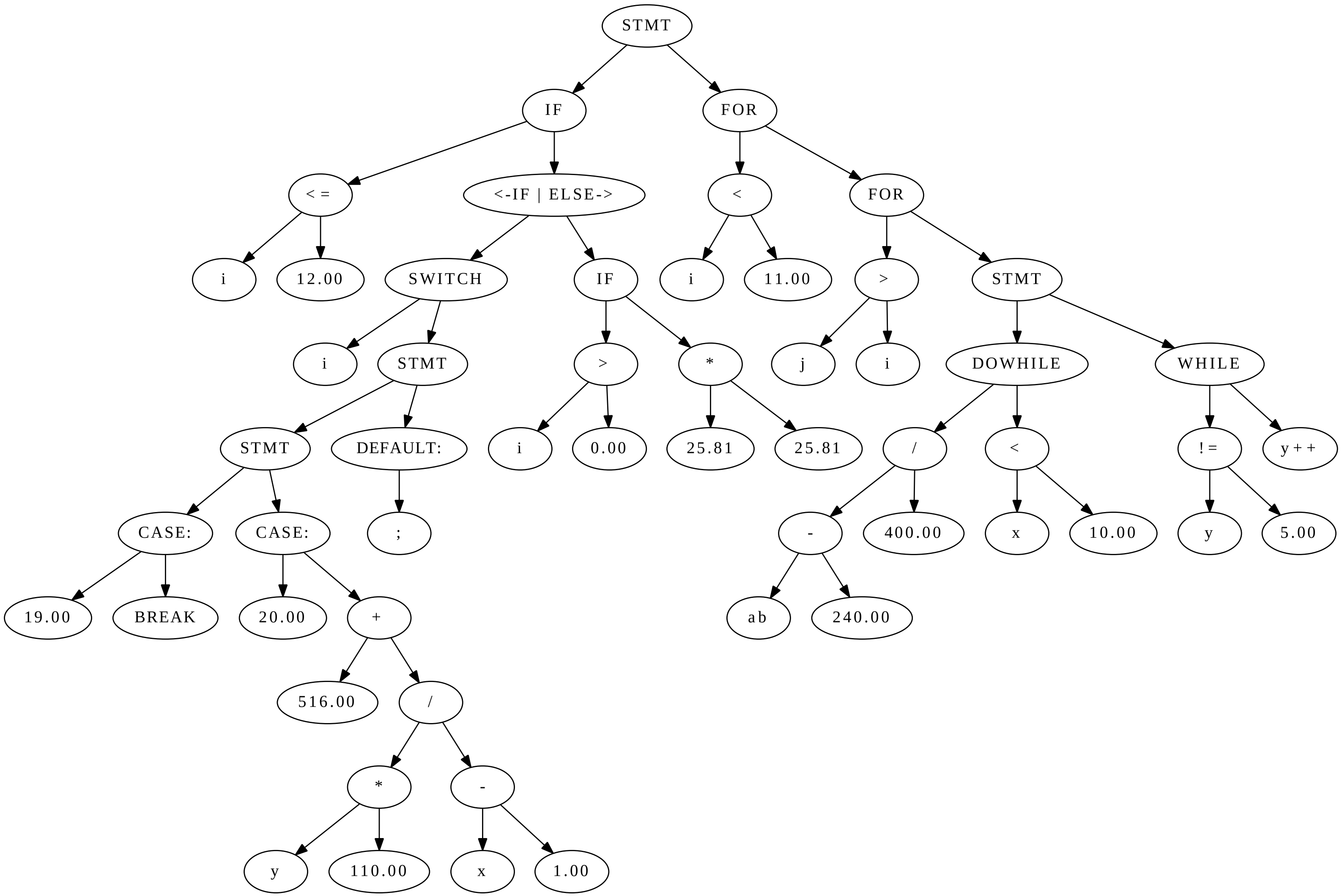

Figure 6: the *AST* for the function *func* in Figure 5.

References

[1] John Levine and Levine John. *Flex & Bison*. O'Reilly Media, Inc., 1st edition, 2009.

[2] Jim Mahoney. Bison calculator. https://github.com/jimmahoney/bison_calculator. Accessed: 03.05.2019.