

Tree++

A Tree Manipulation Language

By:

Allison Costa
Manager
Arc2211

Laura Matos
Tester
lm3081

Jacob Penn
System Architect
jp3666

Laura Smerling
Language Guru
les2206

Table of Contents

Table of Contents	2
Introduction	4
Abstract	4
Motivation	4
Problems	5
Language Tutorial	6
Basic Syntax	6
Simple Examples to Illustrate Syntax	6
Language Manual: Tree++	8
Project Plan	20
Planning Process	20
Development Process	21
Project Timeline	22
Software Development Environment	23
Style Guide	23
Roles and Responsibilities	23
Allison Costa: Manager	23
Laura Matos: Tester	24
Jacob Penn: System Architect	24
Laura Smerling: Language Guru	24
Project Log	24
Architectural Design	25
Components	25
Interfaces Between Components	25
Test Plan	26
Unit Testing	26
Regression Testing	27
Tests in C	27
Tests to Decl Branch	28
Lessons Learned	29
Allison Costa	29
Laura Matos	29

Jacob Penn	29
Laura Smerling	29
Advice	30
Appendix: Tree++	30
Appendix: MicroC+	68
Appendix: MicroC+ C-Code:	68
Appendix: MicroC+ Code:	92
Appendix: Original Tree++ Code	126
Appendix: Other Info	139
Project Log Based On GitHub	139

Introduction

Abstract

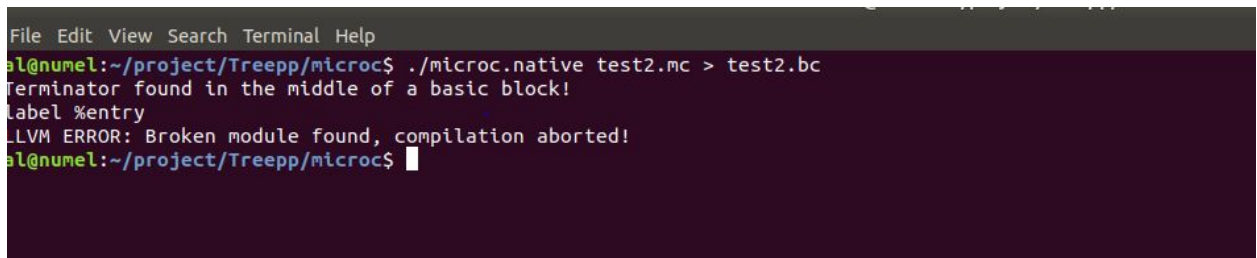
Tree algorithms are relevant to AI, CS theory and data manipulation. Tree++ is a language that uses the underlying data structure of a tree in order to simplify complex tree based algorithms. Like in most languages, the trees can be used for a number of sorting and search algorithms, such as Breadth First Search or Binary Search; however, in the Tree++ structure, trees can also be used for applications such as managing temporary results, which in other languages is typically allocated to the stack data structure. By limiting the user to trees, the hope is to allow programmers to realize their efficiency and effectiveness, even when implementing programs more typically coded with other data structures (such as lists, graphs, queues, stacks, etc). Narrowing the scope of the language to trees also provides a quick and intuitive way to manipulate data. Often new programmers are wary or confused by the wide a way of data structures, not all of which are very straightforward, thus Tree++ helps narrow the focus and expand the typically scope of trees in other languages such as Java, C, and C++. Overall, our implementation increases the efficiency and intuitiveness of typical tree-based algorithms and can be effectively used to program other algorithms, which are usually allocated to different data structures.

Motivation

The motivation for our language arose from our confidence in the tree data structure to effectively solve complex and interesting algorithms and yet our frustration with the current set up of trees in Java and C where search and sorting algorithms force the user to utilize multiple loops and traverse through the tree in a fairly inefficient manner. We plan to both allow the user to decide a great deal about the layout and balancing of the tree; however, we also want to provide general structural information about child, parent, and level of node, which will prove useful when implementing search and sorting algorithms. These features make our language ideal for performing a number of programs including and beyond those typically allocated to trees in object-based programming languages. Like in most languages, our trees can be used for a number of sorting and search algorithms, such as Breadth First Search or Binary Search; however, our tree structure can also be used for applications such as managing temporary results, which in other languages is typically allocated to the stack data structure. Overall, our implementation increases the efficiency and intuitiveness of typical tree-based algorithms and can be effectively used to program other algorithms, which are usually allocated to different data structures.

Problems

The major problem with Tree++ is that it does not create main correctly. This was partially because of our design decisions against creating main in the typically way. We wanted inline declaration and a hidden main that would not force the user to create main for every program. In order to do this, we decided (partially because of seeing the successful implementation in BURGer and PLTree) to pass all of our functions as statements, then check to make sure no functions were called main and create our own hidden main in codegen. Although it is technically possible, we have since learned that it is far better to create main in either ast or semant so that the generated file of semant has a main, rather than having to go back into functions and add the main in codegen after the code has already been semantically checked. This created a major error in our code (shown below), which would not let us implement the basic blocks required for LLVM. Ideally, we would have been able to fix this problem and change our code so that the ast was wrapped in a main; however, our original implementation was too entrenched to change without completely rewriting most of our codegen and semant, which we did not have time for once we realized the severity and consequences of our design decision.



```
File Edit View Search Terminal Help
al@numel:~/project/Treapp/microc$ ./microc.native test2.mc > test2.bc
Terminator found in the middle of a basic block!
Label %entry
LLVM ERROR: Broken module found, compilation aborted!
al@numel:~/project/Treapp/microc$
```

When we tried to put some of our implementation into past project's stable code, we encountered many more errors. This emphasized how unique each person's code is and how difficult it is to translate between languages; however, it also made it very difficult to demonstrate the working code we had despite our language's broader problems. We finally decided to go back to a very early implementation of MicroC, before it was greatly modified by our group and try to show some of the very basics we learned/programed through this process, including some of the interaction with the C-code that controls nodes.

After presenting to Professor Edwards, he gave us some suggestions on how to fix the problem that was breaking our basic blocks. We went back to our Tree++ code and were able to fix quite a few of the issues. Our main is still hidden in codegen; however, it now allows Tree++ to compile and display code. All of our types besides string work, our operators, if statements and blocks, inline assignment and declaration, and functions all work. While we would have loved to add more features, especially including nodes, we were unable due to time constraints.

Language Tutorial

Basic Syntax

Our program uses a mix of C and Python-esque syntax. Our language supports types int, float, bool, string, void. Nodes can be of any of these specific types, but nodes must be of the same type as their children and ancestors. All types (including nodes) allow for inline declaration and assignment. Code should not be need to be wrapped in a main function, but rather supports code outside of functions as well as function declaration.

Simple Examples to Illustrate Syntax

Control Flow:

```
node<string> hello_world = ("root");
hello_world.root;
...
node<string> n = ("hello");
hello_world.add_child(n);
node<string> m = ("world");
hello_world.add_child(m);
printn(hello_world);
int x = 0;
while(x<1){
hello_world <<; /* shifts the child nodes left*/
x = x+1;
}
printn(hello_world);
```

Output: root hello world root world hello

Versus Function Declaration:

```
node<string> h = ("hello");
h.root;
node<string> m = ("world");
```

```

h.add_child(m);
def node<string> rotate(node<string> root, node<string> child ){
    root^child;
    return root;
}
printn(rotate(h));

```

Output: world hello /*the root is now the child and the child is now the root*/

Tree++ Features:

Parser:

```

| "node"    { NODE }
| ".root"   { ROOT }
| ".data"   { DATA }
| ".depth"  { NODE_DEPTH }
| "<<"      { LSHIFT_NODE }
| ">>"      { RSHIFT_NODE }
| "^"       { SWAP_NODE }
| ".add_child" { ADD_CHILD }
| ".delete_node" { DELETE_NODE }

```

C-Functions:

```

void init_root(struct Node *node); // done
struct Node *create_int_node(int data); // done
struct Node *create_char_node(char data); // done
struct Node *create_float_node(float data); // done
void delete_node(struct Node *node); // done
void add_child(struct Node *parent, struct Node *child); // done
void deep_swap(struct Node *node_a, struct Node *node_b); // done
void shift_left(int index, struct Node *child); // done
void shift_right(int index, struct Node *child); // done
int is_root(struct Node *node); // done
int is_empty(struct Node *node); // done
void add_child(struct Node *parent, struct Node *child); // done
int is_root(struct Node *node); // done

```

```
int is_empty(struct Node *node); // done
int get_depth(struct Node *node); // done
struct Node *get_root(struct Node *node); // done
```

Language Manual: Tree++

Lexical Conventions

If a sequence of characters can be split into tokens ambiguously, the one with the longest first token will be adopted.

Whitespace

Whitespace consists of spaces, tabs, and newline characters. Whitespace characters separate tokens, but are otherwise ignored by the compiler.

Comments

Comments begin with the character combination (** and end with **). Any text in between the beginning and end of a comment is ignored by the compiler. Comments can be nested and multiline.

Identifiers

An identifier is any sequence of letters, numbers, or the underscore character `_`. Identifiers are case sensitive and must not begin with a numerical digit.

Literals

Boolean Literals

A boolean literal can take on the value `true` or `false`.

Integer Literals

An integer literal is a sequence of digits, representing a number in base 10.

Floating-point Literals

A floating-point literal consists of two sequences of digits separated by `.`, a decimal point. The sequence to the left of the decimal is the integral component, while the sequence to the right is the fractional component. The integral component must not be empty but the fractional component may.

?String Literals

Any sequence of characters surrounded by double quotes (excluding the `"` double quotation character) constitutes a string literal. Certain characters may also be expressed with an escape sequence:

\n newline character
\\" double quote
\' single quote
\t tab

Keywords

The following is a list of reserved keywords and may not be used otherwise:

int void true
false bool if else
float for while parent
func return empty node<type>
print NULL data

Data Types

Tree++ uses a variety of standard data types from C, with an addition of type that is mathematically expressive, such as node. Functions are first-class objects in Tree++, so a function type (func) is also present. The operators that can be used with each datatype are explained in the next section on expressions and operators.

int

We support a 32-bit integer data type, int. It might be declared as. All declarations are inline and assignments are inline.

int bar = 5;

float

The float type is a floating-point data type used to store real values. It might be declared as:

float bar = 5.5;

bool

The bool type is a boolean type taking on two values, true and false. It might be declared as:

bool bar = false;

(*string

The string type is a type used to store and manipulate strings of ASCII characters. It might be declared as:

*string bar = "Hello World!"; *)*

void

The void data type represents an empty type, behaving just as it does in C.

function

Functions in Tree++ are first-class objects: they can be passed as arguments and returned from other functions. Therefore, Tree++ supplies a function type, `func`. Refer to subsection 9 for more in-depth information on functions.

Expressions and Operators

We describe expression in order of precedence below, starting with those with the highest level of precedence. Any expression can be disambiguated by parentheses. Consider the following example:

`10 * 3 + 4 ; (* 34 *)`

`10 * (3 + 4) ; (* 70 *)`

Also note that each unary and binary operator is only allowed for the data types which have it defined (definitions of various operators can be found in the previous subsection).

Primary expressions

Identifiers:

Identifiers are named variables including function, and they have a strict type assigned at declaration.

Literals The literals are as specified in the Lexical Convention subsection.

Negation Operators:

For an expression, `expr`, which is of type `int` or `float`, `(-expr)` returns the negated value. On the other hand if `expr` is of type `bool`, `(!expr)` returns the negated value.

Assignment Operators

Variables are assigned using the `=` operator. The left hand side must be an identifier while the right hand side must be a value or another identifier. The LHS and RHS must have the same type, as conversions or promotions are not supported. The variable assignment operator groups right-to-left.

In the format below, `l_val` is any identifier and `r_val` is any expression. The following assignment operators are right associative.

Assigns `r_val` to `l_val` then the entire expression takes on the new value of `l_val`:

`(l_val = r_val)`

Computes `l_val * r_val` and assigns it to `l_val` then the entire expression takes on the new value of `l_val`:

`(l_val *= r_val)`

Computes `l_val / r_val` and assigns it to `l_val` then the entire expression takes on the new value of `l_val`:

`(l_val /= r_val)`

Computes `l_val % r_val` and assigns it to `l_val` then the entire expression takes on the new value of `l_val`:

`(l_val %= r_val)`

Computes `l_val + r_val` and assigns it to `l_val` then the entire expression takes on the new value of `l_val`:

`(l_val += r_val)`

Computes `l_val - r_val` and assigns it to `l_val` then the entire expression takes on the new value of `l_val`:

(l_val -= r_val)

Multiplicative Operators:

The multiplicative operations are left associative and the return value matches the type of the operands. The operands must match type. The * and / operations exist for int and float types, while the % operation only exists for int.

Computes $expr1 * expr2$ and the entire expression takes on the resulting value:

*(expr1 * expr2)*

Computes $expr1 / expr2$ and the entire expression takes on the resulting value:

(expr1 / expr2)

Computes $expr1 \% expr2$ and the entire expression takes on the resulting value:

(expr1 \% expr2)

Additive Operators :

Additive operators are left associative and the return value matches the type of the operands. The operands must match type. The + operation exists for int, float, and string (concatenation) types while the - operation only exists for int and float types.

(expr1 + expr2)

Computes $expr1 + expr2$ and the entire expression takes on the resulting value.

(expr1 - expr2)

Computes $expr1 - expr2$ and the entire expression takes on the resulting value.

Relational Operators:

The relational operators are left associative and evaluate to a boolean. These operations exist for int and float types.

Compares $expr1$ to $expr2$ and if $expr1$ is strictly less than $expr2$ it returns true; otherwise false:

(expr1 < expr2)

Compares $expr1$ to $expr2$ and if $expr1$ is strictly greater than $expr2$ it returns true; otherwise false:

(expr1 > expr2)

Compares $expr1$ to $expr2$ and if $expr1$ is less than or equal to $expr2$ it returns true; otherwise false:

(expr1 <= expr2)

Compares $expr1$ to $expr2$ and if $expr1$ is greater than or equal to $expr2$ it returns true; otherwise false:

(expr1 >= expr2)

Equality Operators:

The equality operators are left associative and evaluate to a boolean. These operators exist with any type, and can also be used with NULL to test whether a variable is null. For strings, == is structural (lexicographical) equality. For nodes, == is physical equality.

Compares $expr1$ and $expr2$ and returns true if the two expressions have the same value:

(expr1 == expr2)

Compares $expr1$ and $expr2$ and returns true if the two expressions do not have the same value:

(expr1 != expr2)

Logical Operators

Logical operators on booleans are left associative and evaluate to a boolean. Returns false if *expr1* and *expr2* are both false; otherwise returns true:

(expr1 && expr2)

Returns true if *expr1* and *expr2* are both true; otherwise returns false:

(expr1 || expr2)

Function Expression

In Tree++, functions are treated as a kind of expression. We allow for inplace anonymous functions as follows:

```
function type name(arg1, arg2,...){
    statement1;
    statement2;...
}
```

The arguments, *arg*, are of the form *type identifier* indicating the type and local name of the function's arguments. The statements, *statement1*, *statement2*... are specified in the next subsection and ends in a return statement returning a value of the specified return type. For more information, refer to the section dedicated to functions.

Function Calls

A call to a function is an expression that takes on the same value as the return value of the function called with the given arguments. For example:

```
func add<int>(int a,int b) {
    return a+b;
};
print(add(2, 4)); (* 6 *)
```

Statements

Expression statements

A statement is defined as any expression followed by a semi-colon. Below are several examples:

31;

l == (3-2);

x++;

y = 10;

The first two statements are evaluated but are generally not useful since they have no side effect. On the other hand, the last two will alter the variables x and y respectively.

Control Flow

Conditionals

If-else statements use the following formats:

```
if (condition) {statements}
```

```
if (condition) {statements} else {statements}
```

Each condition is evaluated to either true or false, and the corresponding set of statements is executed accordingly. The braces are optional if there is only one statement; a sequence of statements must be enclosed within corresponding braces.

Loops

C-style while loops and for loops are provided, such as the following:

```
while (condition) {statements}
```

```
for (initialization; condition; update) {statements}
```

Block Statement

A block statement is a list of zero or more statements, enclosed within corresponding braces.

```
{  
int x;  
x = 5;  
}
```

As in C, blocks are often used as the body in conditional statements or loops and can be nested inside a bigger block. See subsection 8 for more information on rules on scope related to block statements.

```
{  
    int x = 5;  
  
    {  
int x = 2;  
x == 5; (* this evaluates to false *)  
    }  
x == 5; (* this evaluates to true *)  
}
```

Return statement

Return statements take the form of:

```
return expression;
```

The return expression can be any single expression or omitted. The type of the expression must match the return type of its enclosing function.

Functions

Function definition

The general form for function definition is:

```
func function-name<return_type>(arguments) {statements, return-type return};
```

where return-type can be of any data type discussed above. If no value is returned from the function, then one can use the return type void. Arguments are separated by comma, with each having its own type and identifier. The number of arguments can be zero or more. An example of a function definition is:

```
function int add(int a, int b)
{
return( a + b);
}
```

Note that return-type and type of arguments need to be given.

Calling functions

A function can be called using its name and any necessary parameters. The number and type of parameters must match with the function definition. As with function declaration, function parameters are separated by a comma. Continuing with the example above, we can call add as follows:

```
add(3, 6); (* this will return 9 *)
```

Program Structure and Scope

A Tree++ program must exist within a single source file. As with scripting languages, Tree++ programs start execution from the beginning of the file.

Scope

Tree++ follows scoping rules similar to those in C. Variables declared in the outer body of the program (i.e., not within any enclosing set of braces) are considered global and can be accessed throughout the entire program, but not outside the file in which they reside. Local variables can only be accessed within the block in which they were declared, and will shadow any existing variables with the same names in the outer scope. Formal variables of functions are only visible within their corresponding functions.

A declaration is not visible to statements that precede it. For example:

```
int x = y; (* error: y is not yet declared at this point *)
int y = 6;
```

Built-in Functions

We include the following built-in functions:

Printing

The print function prints the formal argument (either a string, int, or float) to standard output.

```
printbig("hello"); (* prints "HELLO" followed by a newline *)
print(5); (* prints 5 followed by a newline *)
print(3.5); (* prints 3.5 followed by a newline *)
```

TODO:

The rest of the LRM includes code for nodes that we have written and partially implemented with MicroC+, but were unable to integrate properly with Tree++. As a result, for our final turned in assignment, we decided to remove Node, but some of the node functions are still visible throughout Tree++, as we fully extended to incorporate it once our code started to work.

Enhanced for loops

They can either be followed by a single statement to be looped, or by a sequence of statements enclosed within brackets. Tree iteration over nodes and children is also allowed, using "for each" loops, which take the following format:

for value in range(aug1, aug2){statement} - Iterates through the nodes based on the bfs numbers in the tree

for value in newNode{statement} - Iterates through all of the nodes in the entire tree according to BFS

for x in functionOrder() range(aug1, aug2){statement } - One can write their own iterative function if they desire with their desired order and iterate through this function through the for loop

for x in functionOrder() newNode{statement } - One can write their own iterative function if they desire to iterate through all the nodes in the entire tree

The braces are optional if there is only one statement; a sequence of statements must be enclosed within corresponding braces.

node

The node data type is a generic types that can be used in the following manner:

```
Node<type> foo;
```

where foo can only hold values of type T. A node consists of parent, a list of children and a data value.

8 Operators

Node Operators:

Node operators are very important for shifting the information from one direction to another. A node operator used multiple times groups left to right. Combining different node operators in one statement must have the parent node first and the preceding children second. The node operators are left associative when they are stacked. Complete the operation to the left first and then continue to the preceding operations. Expr1 is the parent expression and Expr2 is the child.

- ^

Deep swap node and node (can be parent and child or child and child)

- o If you deep swap node and parent then the child becomes the new parent node and the parent becomes a child of the original child. All children of the original parent remain attached to the parent, which is now the new child of the original child.
- o This is present to write percolating logic easily
- o `expr1 ^`
 - The first child is swapped with the parent
- `<<`
 - o Shift children left
 - o If you want to specify the exact number of children to shift left you can add an index in [] at the end of the operators used. The index is counted from left to right starting at 0.
 - o `expr1<<`
 - If the children are shifted without an addition or subtraction than the first child becomes the last child and the last child becomes the second to last child.
 - If the index is out of range for the child, then an error is printed
- `>>`
 - o Shift children right
 - o If you want to specify the exact number of children to shift right you can add an index in [] at the end of the operators used. The index is counted from right to left starting at 0.
 - o `expr1>>`
 - If the children are shifted without an addition or subtraction than the first child becomes the second child and the last child becomes the first child.
 - If the index is out of range for the child, then an error is printed

Shift and swap operators explained:

```

  a
 / \
b   c

```

`expr1` is the parent, `expr2` is the input:

`expr1 <<+expr2` shift children left and add node in tree last

```

char new_data = 'd';
root(newNode)<<+ new_data;

```

```

  a
 // \
b c  d

```

`expr1 >>+ expr2` shift children right and add node in tree first

```

char new_data2 = 'e';
root(newNode)>>+ new_data2;

```

```

  a
 /\ \ \
e b c d

```


expr1 <<- shift left and delete last node in tree

```
root(newNode)<<-;
```

```
  a
 / \ \
e b c
```

expr2 >>- shift right and delete first node in tree

```
root(newNode)>>-;
```

```
  a
 / \
b c
```

expr1 <<^ shift left and swap parent node and child in tree

```
root(newNode)<<^;
```

```
  c
 /
  a
 /
  b
```

expr1 >>^ shift right and swap parent node and child in tree

```
root(newNode)>>^
```

```
  c
 |
  a
 /
  b
```

- If there is no node to the left to swap, the only remaining child will swap with the parent

```
  a
 / \
c b
```

- If there are no children then the program will throw an error when one tries to shift

expr1 is the parent, expr2 is the input, expr3 is the counter to specify exact location of transition:

expr1 <<<+ [expr3] expr2 shift children left and add node in tree last

expr1 >>>+ [expr3] expr2 shift children right and add node in tree first

expr1 <<- [expr3] shift left and delete last node in tree

expr2 >>-[expr3] shift right and delete first node in tree

expr1 <<^[expr3] shift left and swap parent node and child in tree

expr1 >>^[expr3] shift right and swap parent node and child in tree

If expr3 is not present in the tree then the program will throw an error

Node Literal

A node can be expressed as [expr1, expr2] where expr1 corresponds to the parent of the node and expr2 corresponds to the data. There are no declarations without assignment.

```
node<expr1> newNode = (expr2, expr3);
```

4 Manipulation of Nodes

Built-ins

We provide built-in operations in order to allow for easy manipulation of the node type:

Node<type> *newNode* = (*data*) Declares a new node with parent and data

newNode.parent Returns the parent node

newNode.data Returns the data stored in the node

newNode:n Returns the nth node of the tree according to BFS ordering

The head of the tree is 0, the first child is 1 ...n If there are no nodes in position n the language will throw an error

newNode:n(newNode:i) Returns the grandchild in position i according to bfs where the head is reset to position n. This allows index access of sub-trees.

5 Node Internals

5.1 Mutability and References

Nodes are mutable data types, so the members of an instance of each type (*node.parent*, *node.data*) can be updated. This allows for nodes to be easily manipulated after construction.

Node variables are treated as data values. Therefore, one variable must refer to the same node.

Assignments, for instance, will act as tree copying and data swapping:

```
node<string> newNode = ("hello");
```

```
node<string> secondNode = newNode;
```

```
newNode = ("test");
```

```
print(newNode.data); (*test*)
```

```
print(secondNode.data); (*hello*)
```

All behavior must be clearly defines as the results will change for every assignment. Additionally this only works if the nodes are the same type as the type is specified during declaration.

5.2 Indices

Integer indices are used to identify location of nodes with in trees based on the head of the node. The ordering of the nodes is BFS where the head node is 0 and the first child is 1. If a new child is added in

the center of the tree the indices will shift. Nodes are not added based on index they are added by referencing parents and shifting children. Consider the example below:

```
print(newNode.data); (*c*)
```

```
  a
 / \
b   c
char in = 'd';
newNode.parent >> + in;
print(newNode:2.data); (*b*)
  a
 / / \
d b  c
```

5.3 Invalid Nodes

When a node is added to a tree with a parent that does not correspond to a node in the tree, an error will occur. For example:

```
  a
 / \
b   c

node<char> newNode = [newNode:3, 'e'];
//error this is an invalid parent argument, check your index
```

Hence, it is recommended to make sure that a tree contains all necessary parent nodes before adding a new child node.

5.4 Adjacency Linked Lists

Adjacency linked lists are internally updated within a tree. Each time a child is added or removed, the adjacency linked lists of the corresponding children will be updated accordingly.

5.5 Removing Parent Nodes from a tree

When a parent node is removed from a tree, all children involving that node will also be removed from the tree.

Node functions

The built-in node functions are explained in more detail under the subsection of the manual on the graph datatype. They are listed below:

```
empty(node)
height(node)
```

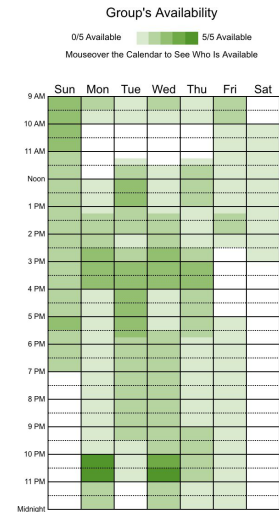
depth(node)
root(node)

Project Plan

style guide

Planning Process

Our team met once a week on Thursdays from 2:30-4:00pm to work on the project. Our meeting schedule with Justin, our TA, was unfortunately much more erratic. At the beginning of the semester we met once a week on Wednesdays from 7:30-8:30/9:00pm. This worked well because it gave us all time to work on the code and figure out what questions we still had by our meeting the next day. By mid October, a number of obligations and commitments from each person in the group and some health trouble with one group member, made meeting much more difficult and our original time impossible. Allison, as the manager of the group, sent out a When2Meet to try to find a time that worked for everyone. With such limited results, it was agreed that Wednesday from 10:30-11:30pm would be our meeting time. Unfortunately this was also not the most consistent time and so we also met on multiple Sundays (both just as a group and with Justin) especially during November/December.



I think this struggle to find times that worked for everyone was particularly detrimental to our progress. It also took us a while to truly understand what it meant to code in vertical slices. We first broke up the assignment based on files (two of us working on codegen and two on semantic). This was not productive to testing because you could not test until everyone had pushed their separate parts to GitHub. Without testing across programs, our progress felt like it was going quite well when in actuality there were many inconsistencies and bugs when integrating the full program. Thus despite the fact each group member spent roughly 3-5 hours on this project per week (with some inconsistencies of course), our progress as a team was quite slow. We later decided to code by language concept instead of by coding concept, so everyone was forced to become fairly familiar with all parts of the code, but you could also test on your own branch independently from any other updates. The sped coding up quite a bit, but also made it difficult to help one another when people became more experienced in very neash sections of the code.

Ultimately, our planning process failed, as we were unable to generate a complete project. While we hope this documentation helps illuminate the amount of work we put into the project, we clearly did not allocate enough time. While all of us are quite independently busy and we are

all guilty of not better handling that work load, I do believe our group's trouble with finding time to meet up and work together or with guidance from the TA greatly contributed.

Development Process

When we began concept and code development, we would tend to meet up to discuss specific problems or questions, but we would usually code independently. I believe our progress was on track throughout the turn in deadlines of our proposal, language reference manual, and LRM/Parser. Unfortunately, once MicroC was released, it took us a while to realize we were supposed to create our code off of MicroC. Once Justin made it clear this was probably the best way to create this language, we started a new repo with just MicroC code and used what we had already done to update it. In some ways this was useful because MicroC had many things we did not know how to implement yet, but it also was quite challenging starting over. For HelloWorld turn in assignment, we just barely had figured out MicroC and were still struggling to implement some elements that our original independent code had.

I think part of what made this project challenging was that we had very specific ideas about what we wanted our language to do. As a result, we did not have a good language to base our code off of or to see how they implemented things. There are surprisingly few tree languages published throughout the years of PLT and many of them do not have codegen. Although some graph languages (such as Justin's group's workspace or Giraph or yeezyGraph) were fairly useful, it was quite difficult to find a good model. All of these projects and others, such as BURGer and PLTree, were useful but ultimately incredibly different from our code. This led us to have multiple iterations of our project each implementing certain things (primarily functions and nodes) slightly differently. I think because all of these code examples are so different (some use the `sast` in codegen and others do not, some have weighted edges and other `laura` counts, some `cast` and others do not, etc.), it was challenging to come up with a cohesive design idea between all of us.

Ultimately, our Tree++ code has stayed very similar to our original LRM (other than removing tuples for increased simplicity); however, this code did not work correctly until the day of the presentation. There is a major design flaw in how we wrap our system with a main function, leaving this to codegen (as exemplified in the BURGer code). As we were not able to get this code, that more accurately displayed our language concept, to work, we tried a number of approaches to fixing it. Jacob, mainly, stayed trying to work out many of the bugs, whereas Laura S and Allison tried implementing many of our distinct functionalities in the stable environment of BURGer, which is the code we first based many of our implementations off of. When this was also not working, mainly because of BURGer's lack of `sast` and thus a huge re-working of codegen and `semant`, we reverted back to a very early version of our modification of MicroC, which is mainly MicroC but with a few added elements, such as `String` and `Mod`. This MicroC+ version is similar to what we turned in for Hello World, however, it also includes strings and some node implementation that includes the C code backend. However, it does not

accurately reflect the time and progress we feel we have made in coding and understanding how to code a language.

After presenting to Professor Edwards, we were able to make quite a few big changes and get our Tree++ code to work for a number of implementations. Currently, the main is still hidden in codegen; however, it now allows Tree++ to compile and display code. All of our types besides string work, our operators, if statements and blocks, inline assignment and declaration, and functions all work. While we would have loved to add more features, especially including nodes, we were unable due to time constraints. We focused on getting some of these features working and creating test cases for them rather than try to include nodes, which took such a significant amount of time to implement them even partially in MicroC+.

Project Timeline

Dates	Goals
September 8-25	Assemble members of team, create established meeting times, develop idea for language, write proposal
September 26	Submit proposal
September 27-October 15	Work on developing language from conceptual state to having a significant chunk of code written
October 16	Submit LRM and Parser (also included scanner and ast)
October 16-November 11	Work on developing our program with MicroC as reference
November 11	Created new git repo after realizing we were supposed to start with MicroC as our basis code
November 11-14	Work on HelloWorld with MicroC
November 14	Turn in HelloWorld for MicroC + mod
November 15-December 15	Working on Tree++ code: changing from MicroC to our language
December 15-18	Got major error in Tree++ code because of main and started to work on fixing project/adding implementations to first BURGER code and then MicroC from HelloWorld turnin
December 19	Present Final Presentation and submit Final Report

Software Development Environment

To build giraph, we used these languages and development tools:

- OCaml version 4.07.0: for scanning, parsing, and semantic checking
- Illvm version 7.0.0
- C: for building and manipulating nodes
- Makefile: for compiling and linking
- Git and Github for version control and hosting our git repository, respectively
- Bash Shell scripting: for automating testing

Style Guide

We used the following conventions while programming our Tree++ compiler, in order to ensure consistency, readability, and transparency.

- OCaml editing and formatting style to write code for compiler architecture
- C language editing and formatting style for inspiration for Tree++ program code
- C language editing and formatting for the c-integration code
- Comments for the OCaml code should be consistent and explanatory, the structure is file dependent

A few other style guidelines to note:

- File names end in .mc
- The file structure should remain the same as microc to limit confusion
- Variable identifiers begin with a lowercase letter and are camelcase
- Function identifiers begin with a lowercase letter and are camelcase
- Never include a main function in Tree++ program

Some past group's projects that were particularly inspiring:

- Workspace
- Giraph
- BURGer
- PLTree

Roles and Responsibilities

Allison Costa: Manager

My responsibility was to help manage the group, organize us, and keep us on time for deliverables. I did this mostly by facilitating most of the emails and meeting times for our group to meet with Justin (our TA) and to make clear and give reminders about when things were due and what people's different responsibilities were for each section. With the help of my

teammates I also helped code some of the scanner and ast for our initial code as well as some of the first extensions of the MicroC code. I mostly focused on the codegen, sast, and parser.

Laura Matos: Tester

My responsibility was to create tests for the project to run to see if everything is working fine. We had many limiting factors because of the roadblocks we stumbled upon. I also did most of the C backend to provide an API for the project to use to for manipulating nodes. In addition, I created the tests for the C backend, and functions as well as worked vertically to try to iron out problems in in the ast, codegen and parser.

Jacob Penn: System Architect

In this project I was the point person on the semant, and become extremely familiar with what it does as well on how to translate parameters so that code translation works from the ast -> semant -> sast -> codegen. Although this was my main focus, I also made numerous changes to ast, parser, sast, and codegen. I also integrated the c functions into our language.

Laura Smerling: Language Guru

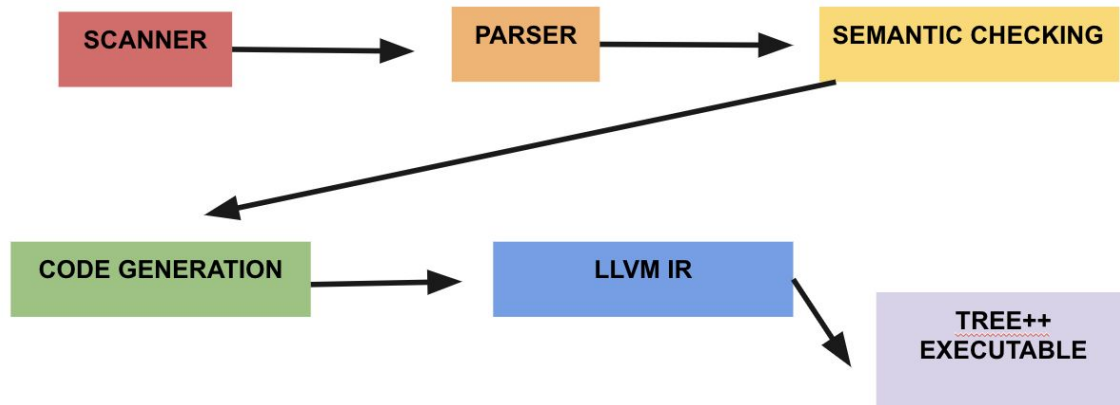
My responsibility was to decide the parameters and specifications for our language. With the help of my teammates I wrote the language reference manual. Throughout the project I modified the manuel to continue to fit our continuously changing program. I then moved on to defining our language in the scanner, parser and ast. After adding some basic features to scanner, parser and ast, as we starting working in vertical slices I moved on to working on the codegen. I mostly focused on adding in the node features to those portions of our project and making sure there there were no shift reduce errors. With the way our project was implemented we broke our connection to node in the codegen and decided to remove it from our final code repository. I also created the sample programs, the power point and the final report.

Project Log

See appendix for our project log.

Architectural Design

Components



Interfaces Between Components

At the beginning of our project we broke the files up by person writing them. With Jacob Penn working on the parser, Allison Costa working on the scanner and Laura Smerling working on the language reference manual. We then decided to work on our language through vertical slices so we all worked on many of the files specifically implementing features.

Scanner.ml (Allison Costa, Laura Smerling, Jacob Penn, Laura Matos)

As having a basis from the micro source file, scanner reads the source Tree++ code and does lexical analysis. We created tokens for all of the different objects that would appear in our input source code. This is the alphabet for our language as it does not allow any illegal characters to pass. If there are illegal tokens then the program will throw a parsing error.

microparse.mly(Allison Costa, Laura Smerling, Jacob Penn, Laura Matos)

The parser reads tokens from the scanner and makes sure that they are syntactically correct. The parser is the spell checker of our language. If the process of parsing has no errors it will successfully create an Abstract Syntax Tree.

Ast.ml (Allison Costa, Laura Smerling, Jacob Penn, Laura Matos)

The abstract syntax tree provides the basic syntax for our language.

Semant.ml(Allison Costa, Jacob Penn)

Our Semantic Checker provides many of the same checks as MicroC. The semantic checker is responsible for walking through the AST that was generated by the parser and make sure that the input file doesn't violate any syntactic rules. Where the parser was able to complain when it found a missing bracket or brace, the semantic checker is able to tell the user when they are doing something that is not supported by the user such as assigning a string literal into a int type. It is also responsible for a table of variable names and functions (symbol table) that we combined into items so that it can complain if a program is trying to access an undeclared variable or function. It also contains a list of all predefined functions in the language and will complain when the parameters don't match in a function call.

codegen.ml(Allison Costa, Laura Smerling, Jacob Penn, Laura Matos)

Codegen "translates" functions and statements given to it from the semantic checker and returns a program that can be read by the toplevel and allows for the implementation of a great deal of our code.

toplevel.ml(Allison Costa, Jacob Penn)

The toplevel of Tree++ is taken almost completely from MicroC. It importantly prints out the LLVM code, but comments out some of the checking.

C_files(Laura Matos)

The backend C code that basically gives the tree functionality to the language. It implements a node structure and provides the API that is integrated/defined in the codegen file.

Test Plan

Unit Testing

Early stages of parser testing were carried out with using ocaml yacc -v and looking through the resulting .output files. We found looking at which states threw errors particularly helpful when first trying to understand how different elements were going to fit into the larger scope of the program (for instance, deciding whether assign should be an expr, a stmt or both, was decided by looking at how the parser implements it). We also ran short programs with the parser trace option ocamlrunparam=p. We continued with unit testing for all of our programs especially when we were delinating work via program rather than idea/component.

Regression Testing

Our test cases main consisted of the tests that MicroC runs automatically with make. These were a general design for any changes to them we made as we tried to implement new functionality. Unfortunately, due to not getting our code working correctly, we have very limited testing, as the vast majority of our work was focused on getting all of our files to compile together without any errors.

We were able to make a couple of tests for the final MicroC+ code we presented, but our testing for Tree++ is incredibly limited, which (in retrospect) should have made us more aware of the danger of not integrating all of our code sooner.

Tests in C

Unlike the tests for the rest of the project, there was a separate directory for the C code in the backend. We kept the tests exclusive to that branch to be compiled and run through a Makefile so there is little confusion. The tests for this section of the repository are unit tests since we were not able to get far enough to link the C backend code with the rest of the project for regression testing. The tests for part of the project could be more verbose since they were unit tests and they demonstrated whether tests for the tree structure were an issue could not reach the end of the tree.

```
-----create_int_node-----
Level: 0      Data: 5
Level: 0      Data: 11

-----create_char_node-----
Level: 0      Data: c
Level: 0      Data: a

-----create_float_node-----
Level: 0      Data: 5.500000
Level: 0      Data: 11.500000

-----delete_node-----
Level: 0      Data: 5
Level: 1      Data: 11
Level: 1      Data: 77
Level: 1      Data: 71
Level: 1      Data: 21
Level: 1      Data: 21

-----init_root-----
is_root 0 == 0
is_root 1 == 1

-----add_child-----
Level: 0      Data: 5
Level: 1      Data: 11
Level: 1      Data: 77
Level: 2      Data: 71
Level: 1      Data: 21

-----deep_swap_same_level-----
Level: 0      Data: 5
Level: 1      Data: 77
Level: 1      Data: 11
Level: 1      Data: 11

Level: 0      Data: 5
Level: 1      Data: 11
Level: 1      Data: 77
Level: 1      Data: 77

Level: 0      Data: 5
Level: 1      Data: 77
Level: 1      Data: 11
Level: 1      Data: 11

-----shift_left_three-----
Level: 0      Data: 5
Level: 1      Data: 11
Level: 1      Data: 77
Level: 1      Data: 71
Level: 1      Data: 21
Level: 1      Data: 21

Level: 0      Data: 5
Level: 1      Data: 21
Level: 1      Data: 77
Level: 1      Data: 71
Level: 1      Data: 11
Level: 1      Data: 21

-----get_root-----
Level: 0      Data: 5
Level: 0      Data: 5
Level: 0      Data: 5

-----is_ancestor-----
Is ancestor 1 == 1
Is ancestor 1 == 1
Is ancestor 1 == 1
Is ancestor 0 == 0
```

Representative Language Programs with Target Language Programs

Source Program (sample_hello_node.mc):

```
int main()
{
    string i;
    i = "hello world!";

    node<string> hello_world = ("root");
    hello_world.root;

    node<string> n = "Hello";
    hello_world.add_child(n);
    node<string> m = ("world");
    hello_world.add_child(m);

    printn(hello_world);
    return 0;
}
```

Unable to show output

Tests to Decl Branch

```
File Edit View Search Terminal Help
/* test2.bc */
int i = 10;
int n = 2;
int z = i/n;
int test = z * i;
~
~
~
~
~
~
File Edit View Search Terminal Help
ModuleID = 'MicroC'
source_filename = "MicroC"
@i = global i32 @
@n = global i32 @
@z = global i32 @
@test = global i32 @
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
declare i32 @print(i8*, ...)
declare i32 @printf(i8*, ...)
declare i32 @printbig(i32, ...)
define i32 @main() {
entry:
    ret i32 @
}
~
~
~/project/Treapp/microc$ ./microc.native test2.mc > test2.bc
Terminator found in the middle of a basic block!
label %entry
LLVM ERROR: Broken module found, compilation aborted!
al@numel:~/project/Treapp/microc$
```

Luckily after presenting and getting some advice from Professor Edwards, we were able to fix at least some of the problems with the decl branch, which is closest to our Tree++ language, but

still unfortunately without a lot of the features we planning and have coded, despite not being able to integrate.

Lessons Learned

Allison Costa

I think I truly gained a new appreciation for how difficult and complex creating a new language is because to make a new language you have to put together parts of multiple different languages. This is a very tricky process and complex to navigate especially when unexperienced in a number of the key languages. I think, especially because my group was stuck on some of our code, I learned a great deal about debugging and editing a language. I also was amazed by how little the unit tests mattered once the code was compiled. While I head Prof. Edwards say this when introducing the project, it did not really sink in until my group was faced with this problem.

Laura Matos

Although the code that works individually is good and usually indicates the code is good, more regression testing is always necessary because passing unit tests leads people to a false sense of security. On the other hand, I've realized that while not being too attached to code helps you with flexibility, creating and abandoning branches every time you run into another error is not a viable way to solve your problems as it leads you to repeat yourself and usually go down avenues that lead to even harder problems and errors to move through.

Jacob Penn

Untested code is broken code. On the other hand, hours of error testing make you a whole lot more familiar with the code in question. I have attained a deep respect for ocaml and the exciting complexity of the language design / compiler process.

Laura Smerling

A huge portion of the project was to learn how to work on a team and be on top of changing schedules. Although each member was working on a different task all of the code needed to integrate properly. This integration I realized is the most important part of the project and took up the most allotment of time. The sooner you integrate the better. Additionally I learned a lot about the way in which programming languages are designed and how compilers are built. I realized that error checking is different when writing a compiler. While unit tests in other cases are helpful they do not register integration.

Advice

Start early and do not be afraid to ask questions until you really understand what is going on in the code and what advice the TAs/Professor is giving you to fix it. It is much better to ask lots of questions and then feel confident in your work, than not ask questions and have more questions later when it is harder to change things.

Appendix: Tree++

Scanner.mll

```
(* Ocamllex scanner for  
MicroC *)
```

```
{ open Microcparse }
```

```
let digit = ['0' - '9']
```

```
let digits = digit+
```

```
rule token = parse
```

```
[' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
```

```
| "/" * "    { comment lexbuf }          (* Comments *)
```

```
| '('      { LPAREN }
```

```
| ')'      { RPAREN }
```

```
| '{'      { LBRACE }
```

```
| '}'      { RBRACE }
```

```
| '['      { LBRACK }
```

```
| ']'      { RBRACK }
```

```
| ';'      { SEMI }
```

```
| ':'      { COLON }
```

```
| ','      { COMMA }
```

```
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '%'      { MOD }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "then"   { THEN }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "string" { STRING }
| "int"    { INT }
| "bool"   { BOOL }
| "float"  { FLOAT }
| "void"   { VOID }
| "node"   { NODE }
| ".data"  { DATA }
| ".parent" { PARENT }
| ".level" { NODE_LEVEL }
```

```

| "<<"    { LSHIFT_NODE }
| ">>"    { RSHIFT_NODE }
| "^"     { SWAP_NODE   }
| ".find_node" { FIND_NODE }
| ".add_node" { ADD_NODE  }
| ".remove_node" { REMOVE_NODE }
| "bfs"     { BFS       }
| "dfs"     { DFS       }
| "function" { FUN      }
| "true"    { BLIT(true) }
| "false"   { BLIT(false) }
| digits as lxm { LITERAL(int_of_string lxm) }
| digits '.' digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm
{ FLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_' ]* as lxm
{ ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^
Char.escaped char)) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }

```

Semant.ml

(* Semantic Checker for the
Tree++ Programming Language

PLT Fall 2018

Authors:

Allison Costa

Laura Matos

Laura Smerling

Jacob Penn

*)

```
open Ast
```

```
open Sast
```

```
module StringMap = Map.Make(String)
```

```
(* Semantic checking of a program. Returns void if  
successful,  
throws an exception if something is wrong.  
Check each global variable, then check each function *)
```

```
let check_program program =
```

```
(* Raise an exception if the given list has a duplicate *)
```

```
let report_duplicate exceptf list =  
  let rec helper = function  
    n1 :: n2 :: _ when n1 = n2 -> raise (Failure  
(exceptf n1))  
  | _ :: t -> helper t  
  | [] -> ()  
  in helper (List.sort compare list)  
in
```

```
(* figure out which items are statements and make a list  
of statements *)
```

```
let stmt_list =
```

```

let stmts_as_items =
  List.filter (fun x -> match x with
    Ast.Stmt(x) -> true
    | _ -> false) program
in List.map (fun x -> match x with
  Ast.Stmt(x) -> x
  | _ -> failwith "stmt casting didn't work")
stmts_as_items
in

(* after you figure out which items are statements, you
need to go through the statements
and figure out which ones contain the variable
declarations and
variable decl+assignment statements *)

let globals =
  let global_list = List.filter (fun x -> match x with
    Ast.VarDec((_, x), _) -> true
    | _ -> false) stmt_list
in List.map (fun x -> match x with
  Ast.VarDec(x, _) -> x
  | _ -> failwith "not turned into global") global_list
in

let functions =
  let functions_as_items = List.filter (fun x -> match x
with
  Ast.Function(x) -> true
  | _ -> false) program
in
  let all_functions_as_items = functions_as_items

```

```

        in List.map (fun x -> match x with
                    Ast.Function(x) -> x
                    | _ -> failwith "function casting didn't work")
all_functions_as_items
in

(* let function_locals =
   let get_locals_from_fbody fdecl =
     let get_vdecl locals_list stmt = match stmt with
         Ast.VDecl(typ, string) -> (typ, string) ::
locals_list
         | _ -> locals_list
     in
     List.fold_left get_vdecl [] fdecl.Ast.body
   in List.fold_left get_locals_from_fbody (List.hd
functions) (List.tl functions)
in *)

let symbols = List.fold_left (fun var_map (varType,
varName) -> StringMap.add varName varType var_map)
StringMap.empty (globals)
in

let type_of_identifier s =
  try StringMap.find s symbols
  with Not_found -> raise (Failure ("undeclared identifier
" ^ s))
in

(* Raise an exception of the given rvalue type cannot be
assigned to

```

```

the given lvalue type *)
let check_assign lvaluet rvaluet err =
  if lvaluet == rvaluet then lvaluet else raise err
in

(* Raise an exception if a given binding is to a void type
*)
let check_not_void exceptf = function
  (Void, n) -> raise (Failure (exceptf n))
  | _ -> ()
in

let built_in_decls = StringMap.add "println"
  { typ = Void; fname = "println"; formals = []; body =
[] }
  (StringMap.singleton "printbig"
  { typ = Int; fname = "printbig"; formals = [(Int,"x")];
  body = [] })
in

let function_decls = List.fold_left (fun m fd ->
StringMap.add fd.fname fd m)
  built_in_decls functions
in

let function_decl s = try StringMap.find s
function_decls
  with Not_found -> raise (Failure ("unrecognized
function " ^ s))

```

```

in

(*checks to see if any library functions are defined by
user - not allowed *)
let check_function func =
  report_duplicate (fun n -> "duplicate formal " ^ n ^ "
in " ^ func.fname)
    (List.map snd func.formals);

  if List.mem "print" (List.map (fun fd -> fd.fname)
functions)
    then raise (Failure ("function print may not be
defined")) else ();

  if List.mem "println" (List.map (fun fd -> fd.fname)
functions)
    then raise (Failure ("function println may not be
defined")) else ();

  if List.mem "printf" (List.map (fun fd -> fd.fname)
functions)
    then raise (Failure ("function printf may not be
defined")) else ();

  report_duplicate (fun n -> "duplicate function " ^ n)
    (List.map (fun fd -> fd.fname) functions);

  if List.mem "main" (List.map (fun fd -> fd.fname)
functions)
    then raise (Failure ("function main may not be
defined")) else ();

```

```

    List.iter (check_not_void (fun n -> "illegal null formal
" ^ n ^
    " in " ^ func.fname)) func.formals;

(* List.iter (check_not_void (fun n -> "illegal void
local " ^ n ^
    " in " ^ func.fname)) func.locals; *)

(* report_duplicate (fun n -> "duplicate local " ^ n ^ "
in " ^ func.fname)
    (List.map snd func.locals); *)

in

let rec expr = function
    Literal l -> (Int, SLiteral l)
  | Fliteral l -> (Float, SFliteral l)
  | BoolLit l -> (Bool, SBoolLit l)
  | Sliteral l -> (String, SSliteral l)
  | Id s -> (type_of_identifier s, SId s)
  | Assign(var, e) as ex ->
      let lt = type_of_identifier var
      and (rt, e') = expr e in
      let err = Failure("illegal assignement " ^
string_of_typ lt ^ " = " ^
      string_of_typ rt ^ " in " ^ string_of_expr ex)
      in (check_assign lt rt err, SAssign(var, (rt,
e'))))
  | Binop(e1, op, e2) as e -> let (t1, e1') = expr e1 and
(t2,e2') = expr e2 in
      let ty = match op with

```

```

        Add | Sub | Mult | Div | Mod when t1 = Int && t2 =
Int -> Int
        | Equal | Neq when t1 = t2 -> Bool
        | Less | Leq | Greater | Geq when t1 = Int && t2 =
Int -> Bool
        | And | Or when t1 = Bool && t2 = Bool -> Bool
        | _ -> raise (Failure ("illegal binary operator " ^
        string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
        string_of_typ t2 ^ " in " ^ string_of_expr e))
            in
            (ty, SBinop((t1, e1'), op,(t2, e2')))
    | FunCall(fname, actuals) as call -> let fd =
function_decl fname in
    if (fname = "print" || fname = "println")
        then
            let sactuals = List.map (fun e -> expr
e) actuals in (fd.typ,SFunCall(fname, sactuals));
        else
            (if List.length actuals != List.length fd.formals
then
            raise (Failure ("expecting " ^ string_of_int
(List.length fd.formals) ^ " arguments in " ^
string_of_expr call))
            else
            List.iter2 (fun (ft, _) e -> let (t, et) = expr e
in
            ignore (check_assign ft t
(Failure ("illegal actual argument: found "
^ string_of_typ t ^
" ; expected " ^ string_of_typ ft ^ " in " ^
string_of_expr e))))
            fd.formals actuals;
            let sactuals = List.map (fun e -> expr e)
actuals in
            (fd.typ,SFunCall(fname,sactuals))) (* this is
pretty sketch *)

```

```

| Unop(op, e) as ex -> let (t, e') = expr e in
  (match op with
    Neg when t = Int -> (Int, SUnop(op,(t, e')))
    | Not when t = Bool -> (Bool, SUnop(op,(t,
e')))
    | _ -> raise (Failure ("illegal unary operator "
^ string_of_uop op ^
                                string_of_typ t ^ " in " ^
string_of_expr ex)))
  | Noexpr -> (Void,SNoexpr)
in

```

```

let check_bool_expr e = if fst (expr e) != Bool
  then raise (Failure ("expected Boolean expression in " ^
string_of_expr e))
  else expr e
in

```

```

let rec check_stmt s = match s with
  Expr e -> SExpr (expr e)
  | VarDec((t,s),e) -> SVarDec((t,s),expr e)
  | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt
b1, check_stmt b2)
  | For(e1,e2,e3,s)-> SFor(expr e1, expr e2, expr e3,
check_stmt s)
  | While(p, s) -> SWhile(check_bool_expr p,
check_stmt s)
  | Return e -> SReturn(expr e)
  (*
    let (t, e') = expr e in
    if t = func.typ then SReturn (t, e')
    else raise ( Failure ("return gives " ^
string_of_typ t ^ " expected "
^ string_of_typ func.typ ^ " in " ^
string_of_expr e)) *)

```



```

| Seq s1 -> let rec check_seq = function
  [Return _ as x] -> [check_stmt x]
  | Return _ :: _ -> raise (Failure "nothing may
follow a return")
  | Seq s1 :: ss -> check_seq (s1 @ ss)
  | s :: ss -> check_stmt s :: check_seq ss
  | [] -> []
in SSeq(check_seq s1)
in

```

```

let realcheck_functions func =
  {
    styp = func.typ;
    sfname = func.fname;
    sformals = func.formals;
    sbody = (List.map check_stmt func.body);}
in

```

```

(* Check for assignments and duplicate vdecls *)
(* let y = (List.map check_stmt stmt_list) *)
let _ = ignore(List.iter check_function functions) in
let convert x = List.map (fun y -> SStmt(y)) x in
let transmit z = List.map (fun y -> SFunction(y)) z in
(transmit (List.map realcheck_functions
functions),convert(List.map check_stmt stmt_list));

```

```

(* ignore(List.iter check_function functions);
ignore(List.map check_stmt stmt_list);*)
(* List.iter stmt stmt_list; *)

```

```
(*report_duplicate (fun n -> "Duplicate declaration or
assignment for " ^ n) (List.map snd globals);*)
```

Sast.ml

```
(* Semantically-checked Abstract
Syntax Tree and functions for
printing it *)
```

```
open Ast
```

```
type sbind = typ * string
```

```
type sexpr = typ * sx
```

```
and sx =
```

```
  SLiteral of int
```

```
  | SFliteral of string
```

```
  | SBoollit of bool
```

```
  | SSliteral of string
```

```
  | SNodeLit of sexpr * sexpr
```

```
  | SNodeData of sexpr
```

```
  | SNodeParent of sexpr
```

```
  | SNodeLevel of sexpr
```

```
  | SLNodeShift of sexpr
```

```
  | SRNodeShift of sexpr
```

```
  | SSwapNode of sexpr * sexpr
```

```
  | SFindNode of sexpr * sexpr
```

```
  | SAddNode of sexpr * sexpr
```

```
  | SRemoveNode of sexpr * sexpr
```

```
  | SId of string
```

```
  | SBinop of sexpr * op * sexpr
```

```

| SUnop of uop * sexpr
| SAssign of string * sexpr
| SFuncall of string * sexpr list
| SNoexpr

type sstmt =
  SSeq of sstmt list
| SExpr of sexpr
| SReturn of sexpr
| SIf of sexpr * sstmt * sstmt
| SVarDec of sbind * sexpr
| SFor of sexpr * sexpr * sexpr * sstmt
| SWhile of sexpr * sstmt

type sfunc_decl = {
  styp : typ;
  sfname : string;
  sformals : bind list;
  sbody : sstmt list;
}
(*
type SStmt = sstmt

type SFunction = sfunc_decl
*)

type sitem =
  SStmt of sstmt
  | SFunction of sfunc_decl

```

```

type sprogram = sitem list

(*type sprogram = sfunc_decl list * sstmt list*)

(* Pretty-printing functions *)

let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (match e with
    | SLiteral(l) -> string_of_int l
    | SBoolLit(true) -> "true"
    | SBoolLit(false) -> "false"
    | SFliteral(l) -> l
    | SSliteral(l) -> "\"" ^ l ^ "\""
    | SNodeLit(e1, e2) -> string_of_sexpr e1 ^ "," ^
string_of_sexpr e2
    | SNodeData(e) -> string_of_sexpr e ^ ".data"
    | SNodeParent(e) -> string_of_sexpr e ^ ".parent"
    | SNodeLevel(e) -> string_of_sexpr e ^ ".level"
    | SLNodeShift(e) -> string_of_sexpr e ^ "<<"
    | SRNodeShift(e) -> string_of_sexpr e ^ ">>"
    | SSwapNode(e1, e2) -> string_of_sexpr e1 ^ "^" ^
string_of_sexpr e2
    | SFindNode (g,param) -> string_of_sexpr g
      ^ ".find_node(" ^ string_of_sexpr param ^ ")"
    | SAddNode (g,param) -> string_of_sexpr g
      ^ ".add_node(" ^ string_of_sexpr param ^ ")"
    | SRemoveNode (g,param) -> string_of_sexpr g
      ^ ".remove_node(" ^ string_of_sexpr param ^
")"
    | SId(s) -> s
    | SBinop(e1, o, e2) ->

```

```

        string_of_sexpr e1 ^ " " ^ string_of_op o ^ " "
    ^ string_of_sexpr e2
    | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr
e
    | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
    | SFuncall(s,e) -> s ^ "(" ^ String.concat ", "
(List.map string_of_sexpr e) ^ ")"
    | SNoexpr -> ""

) ^ ")"

```

```

let rec string_of_sstmt = function
    SSeq(stmts) ->
        "{\n" ^ String.concat "" (List.map
string_of_sstmt stmts) ^ "}\n"
    | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
    | SReturn(expr) -> "return " ^ string_of_sexpr expr
^ ";\n";
    | SIf(e,s1,s2) -> "If(" ^ string_of_sexpr e ^ ") {"
^ string_of_sstmt s1 ^ "} Else {" ^ string_of_sstmt
s2 ^ "}"
    | SVarDec((t, s),e) -> string_of_typ t ^ " " ^ s ^
" = " ^ string_of_sexpr e
    | SFor(e1, e2, e3, s) ->
        "for (" ^ string_of_sexpr e1 ^ " ; " ^
string_of_sexpr e2 ^ " ; " ^
        string_of_sexpr e3 ^ ") " ^ string_of_sstmt s
    | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^
") " ^ string_of_sstmt s

```

```

let string_of_sfunc_decl func_decl =
    func_decl.sfname ^ "<" ^ string_of_typ
func_decl.styp ^
    ">(" ^ String.concat ", " (List.map snd
func_decl.sformals) ^

```

```

    "){" ^
    String.concat "" (List.map string_of_sstmt
func_decl.sbody) ^
    "}\n"

let string_of_sprogram stmts =
    String.concat " " (List.map string_of_sstmt stmts)
(* String.concat "" (List.map string_of_vdecl
vars) ^ "\n" ^
String.concat "\n" (List.map string_of_sfunc_decl
funcs) *)

```

Microparse.mly

```

/* Ocamlyacc
parser for MicroC
*/

```

```
%{
```

```
open Ast
```

```
%}
```

```
%token SEMI COLON LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA
DATA NODE_LEVEL PLUS MINUS TIMES MOD DIVIDE ASSIGN
```

```
%token NOT EQ NEQ LT LEQ GT GEQ AND OR FUN
```

```
%token RETURN IF THEN ELSE FOR WHILE BFS DFS
```

```
%token INT BOOL FLOAT VOID STRING NODE PARENT
```

```
%token LSHIFT_NODE RSHIFT_NODE SWAP_NODE ADD_NODE REMOVE_NODE
FIND_NODE
```

```
%token <int> LITERAL
```

```
%token <bool> BLIT
```

```
%token <string> ID FLIT
```

```

%token EOF

%start program

%type <Ast.program> program

%nonassoc FUN NOELSE

%nonassoc ELSE SEMI COLON

%right ASSIGN

%left PARENT

%left DATA NODE_LEVEL

%left OR

%left AND

%left EQ NEQ

%left LT GT LEQ GEQ

%left PLUS MINUS

%left TIMES DIVIDE MOD

%right NOT

%left LSHIFT_NODE RSHIFT_NODE ADD_NODE REMOVE_NODE SWAP_NODE
FIND_NODE
%%

program:

    item_list EOF { List.rev $1 }

item_list:

    {}

|    item_list item { $2 :: $1 }

```

item:

```
    stmt { Stmt($1) }  
    | fdecl { Function($1) }
```

typ:

```
    INT { Int }  
    | BOOL { Bool }  
    | FLOAT { Float }  
    | VOID { Void }  
    | STRING { String }  
    | NODE LT typ GT { Node($3) }
```

stmt_list:

```
    stmt { [$1] }  
    | stmt_list stmt { ($2 :: $1) }
```

stmt:

```
    expr SEMI { Expr($1) }  
    | RETURN expr SEMI { Return ($2) }  
    | IF LPAREN expr RPAREN THEN LBRACE stmt_list RBRACE ELSE LBRACE  
stmt_list RBRACE { If($3, Seq($7),Seq($11)) }  
    | typ ID ASSIGN expr SEMI {VarDec(($1, $2),  
$4)}  
    | FOR LPAREN expr SEMI expr SEMI expr RPAREN LBRACE stmt_list RBRACE  
{ For($3, $5, $7, Seq($10)) }  
    | WHILE LPAREN expr RPAREN LBRACE stmt_list RBRACE {  
While($3, Seq($6)) }
```


expr:

```
LITERAL      { Literal($1)      }
| FLIT       { Fliteral($1)     }
| BLIT       { BoolLit($1)     }
| ID         { Id($1)           }
| expr PLUS  expr { Binop($1, Add, $3) }
| expr MINUS expr { Binop($1, Sub, $3) }
| expr TIMES expr { Binop($1, Mult, $3) }
| expr MOD   expr { Binop($1, Mod, $3) }
| expr DIVIDE expr { Binop($1, Div, $3) }
| expr EQ    expr { Binop($1, Equal, $3) }
| expr NEQ   expr { Binop($1, Neq, $3) }
| expr LT    expr { Binop($1, Less, $3) }
| expr LEQ   expr { Binop($1, Leq, $3) }
| expr GT    expr { Binop($1, Greater, $3) }
| expr GEQ   expr { Binop($1, Geq, $3) }
| expr AND   expr { Binop($1, And, $3) }
| expr OR    expr { Binop($1, Or, $3) }
| MINUS expr %prec NOT { Unop(Neg, $2) }
| NOT expr   { Unop(Not, $2) }
| ID ASSIGN expr { Assign($1, $3) }
| ID LPAREN actuals_opt RPAREN { FunCall($1, $3) }
| LPAREN expr RPAREN { $2 }
```

```
/* Node Auxillary*/
```

```
/*
```

```
| expr DATA { NodeData($1) }
| expr DATA ASSIGN expr { Assign(NodeData($1), $4)}
```

```

| expr PARENT ASSIGN expr { Assign(NodeParent($1), $4) }
| expr PARENT { NodeParent($1) }
| LBRACK expr COMMA expr RBRACK { NodeLit($2, $4) }
| expr NODE_LEVEL { NodeLevel($1) }
| expr LSHIFT_NODE { LNodeShift($1)}
| expr RSHIFT_NODE { RNodeShift($1)}
| expr SWAP_NODE expr { SwapNode($1, $3) }
| expr FIND_NODE LPAREN expr RPAREN { FindNode($1, $4) }
| expr ADD_NODE LPAREN expr RPAREN { AddNode($1, $4) }
| expr REMOVE_NODE LPAREN expr RPAREN { RemoveNode($1, $4) }

```

```

*/

```

```

fdecl:

```

```

    FUN typ ID LPAREN formal_opt RPAREN LBRACE stmt_list RBRACE
    { { typ = $2;
      fname = $3;
      formals = $5;
      body = List.rev $8;
    } }

```

```

actuals_opt:

```

```

    /* nothing */ { [] }
    | actuals_list { List.rev $1 }

```

```

actuals_list:

```

```

    expr { [$1] }
    | actuals_list COMMA expr { $3 :: $1 }

```

```

formal_opt:

```

```

        /* nothing */ { [] }
    | formal_list { List.rev $1 }

```

formal_list:

```

    typ ID { [($1,$2)] }
    | formal_list COMMA typ ID { ($3,$4) :: $1 }

```

Microc.ml

(* Top-level of the MicroC compiler:
scan & parse the input,

check the resulting AST and generate an SAST
from it, generate LLVM IR,
and dump the module *)

```
type action = Ast | Sast | LLVM_IR | Compile
```

```
let () =
```

```
    let action = ref Compile in
```

```
    let set_action a () = action := a in
```

```
    let speclist = [
```

```
        ("-a", Arg.Unit (set_action Ast), "Print the  
AST");
```

```
        ("-s", Arg.Unit (set_action Sast), "Print the  
SAST");
```

```
        ("-l", Arg.Unit (set_action LLVM_IR), "Print  
the generated LLVM IR");
```

```
        ("-c", Arg.Unit (set_action Compile),
```

```
            "Check and print the generated LLVM IR  
(default)");
```

```
    ] in
```

```
    let usage_msg = "usage: ./microc.native
```

```
[-a|-s|-l|-c] [file.mc]" in
```

```
    let channel = ref stdin in
```

```

Arg.parse speclist (fun filename -> channel :=
open_in filename) usage_msg;

let lexbuf = Lexing.from_channel !channel in
let ast = Microparse.program Scanner.token
lexbuf in
(* let sast = Semant.check_program ast in*)

let m = Codegen.translate (Semant.check_program
ast)
in Llvml_analysis.assert_valid_module m;

(* match !action with
Ast -> print_string (Ast.string_of_program ast)
| _ -> let sast = Semant.check_program ast in
match !action with
Ast -> ()
| Sast -> print_string
(Sast.string_of_sprogram sast)
| LLVM_IR -> print_string
(Llvm.string_of_llmodule (Codegen.translate sast))
| Compile -> let m = Codegen.translate sast in
Llvml_analysis.assert_valid_module m; *)
print_string (Llvm.string_of_llmodule m);

```

Codegen.ml

```

(*
* Code generation for MicroC Programming Language
* Authors:
*)

module L = Llvm
module A = Ast
open Sast

```

```

module StringMap = Map.Make(String)

let translate (functions,statements) =
  let theprogram = (functions) @ (statements) in
  let context = L.global_context () in
    let the_module = L.create_module context "MicroC"
    and i8_t = L.i8_type context
    and str_t = L.pointer_type (L.i8_type context)
    and i1_t = L.i1_type context
    and i32_t = L.i32_type context
    and float_t = L.double_type context
    and void_t = L.void_type context in

  (* types of variables in BURGer*)
  let ltype_of_ttyp = function
    | A.Int -> i32_t
    | A.String -> str_t
    | A.Bool -> i1_t
    | A.Float -> float_t
    | A.Void -> void_t
  in
  (* isolate list of items that match as statements and then form a list of statements
  *)
  let stmt_list =
    let stmts_as_items =
      List.filter (fun x -> match x with
        | Sstmt(x) -> true
        | _ -> false) theprogram
    in

```

```

in List.map (fun x -> match x with
    SStmt(x) -> x
    | _ -> failwith "stmt casting didn't work") stmts_as_items
in

```

(*after you figure out which items are statements, you need to go through the statements

and figure out which ones contain the variable declarations *)

```

let globals =
    let global_list = List.filter (fun x -> match x with
        SVarDec(x, _) -> true
        | _ -> false) stmt_list
    in List.map (fun x -> match x with
        SVarDec(x, _) -> x
        | _ -> failwith "not turned into global") global_list
in

```

```

let decode x = List.map (fun v -> match v with SStmt(y) -> y) x in

```

(* isolate list of statements that are NOT variable declarations *)

```

let not_globals_list = List.filter (fun x -> match x with
    SVarDec(_,_) -> false
    | _ -> true) (decode statements) in

```

(* from list of items in program, form list of functions from items and build the main function *)

```

let functions =
    (* generating the hidden main function *)
    let sfunc_decl_main = SFunction{
        styp = Int;
        sfname = "main";
        sformals = [];
    }

```

```

        sbody =(* SReturn(Void,SLiteral(0)) ::*) not_globals_list;
    }
in
(* filtering out items that match as functions *)
let functions_as_items = List.filter (fun x -> match x with
    SFunction(x) -> true
  | _ -> false) theprogram
in
let all_functions_as_items = sfunc_decl_main :: functions_as_items
in List.map (fun x -> match x with
    SFunction(x) -> x
  | _ -> failwith "function casting didn't work") all_functions_as_items
in

(* Store the global variables in a string map *)
let global_vars =
let global_var map (t, n) =
    if (ltype_of_typ t = str_t)
    then (
        let init = L.const_null str_t in
        StringMap.add n (L.define_global n init the_module) map
    )
    else (
        let init = L.const_int (ltype_of_typ t) 0
        in StringMap.add n (L.define_global n init the_module) map
    )
in
List.fold_left global_var StringMap.empty globals in

```

```

(* printf() declaration *)
let print_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let print_func = L.declare_function "print" print_t the_module in

let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let printf_func = L.declare_function "printf" printf_t the_module in

let printbig_t = L.var_arg_function_type i32_t [| i32_t |] in
let printbig_func = L.declare_function "printbig" printbig_t the_module in

(* let println_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let println_func = L.declare_function "println" println_t the_module in *)

(* Define each function (arguments and return type) so we can call it *)
let function_decls =
  let function_decl map func_dec =
    let name = func_dec.sfname
    and formal_types = Array.of_list (List.map (fun (t,_) -> ltype_of_typ t)
func_dec.sformals)
    in
    let ftype = L.function_type (ltype_of_typ func_dec.styp) formal_types in
    StringMap.add name (L.define_function name ftype the_module, func_dec) map
  in
  List.fold_left function_decl StringMap.empty functions
in

(* Fill in the body of the given function *)
let build_function_body func_dec =

```



```

let (the_function, _) = StringMap.find func_dec.sfname function_decls in
let builder = L.builder_at_end context (L.entry_block the_function) in

let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder in

let local_vars =
  let add_formal var_map (formal_type, formal_name) param = L.set_value_name
formal_name param;
  let local = L.build_alloca (ltype_of_ttyp formal_type) formal_name builder in
  ignore (L.build_store param local builder);
  StringMap.add formal_name local var_map
in

let add_local map (formal_type, formal_name) =
  let local_var = L.build_alloca (ltype_of_ttyp formal_type) formal_name builder
in
  StringMap.add formal_name local_var map
in

let formals = List.fold_left2 add_formal StringMap.empty func_dec.sformals
(Array.to_list (L.params the_function)) in

let function_locals =
  let get_locals_from_fbody function_body =
    let get_vardec locals_list stmt = match stmt with
      SVarDec((typ, string), _) -> if (func_dec.sfname = "main")
    then

```

```

        locals_list
    else
        (typ, string) :: locals_list
    | _ -> locals_list
in
    List.fold_left get_vardec [] function_body
in get_locals_from_fbody func_dec.sbody
in List.fold_left add_local formals function_locals
in

let lookup n = try StringMap.find n local_vars
                with Not_found -> StringMap.find n global_vars
in

(* generate code for different kinds of expressions *)
let rec expr builder ((_, e) : sexpr) = match e with
  SLiteral i   -> L.const_int i32_t i
  | SBoolLit b -> L.const_int i1_t (if b then 1 else 0)
  | SFliteral l -> L.const_float_of_string float_t l
  | SNoexpr     -> L.const_int i32_t 0
  | SId s       -> L.build_load (lookup s) s builder
  | SBinop ((A.Float,_) as e1, op, e2) ->
    let e1' = expr builder e1
      and e2' = expr builder e2 in
      (match op with
        A.Add     -> L.build_add
        | A.Sub    -> L.build_sub
        | A.Mult   -> L.build_mul
        | A.Div    -> L.build_sdiv

```

```

    | A.Mod      -> L.build_srem
    | A.And      -> L.build_and
    | A.Or       -> L.build_or
    | A.Equal    -> L.build_icmp L.Icmp.Eq
    | A.Neq     -> L.build_icmp L.Icmp.Ne
    | A.Less     -> L.build_icmp L.Icmp.Slt
    | A.Leq     -> L.build_icmp L.Icmp.Sle
    | A.Greater  -> L.build_icmp L.Icmp.Sgt
    | A.Geq     -> L.build_icmp L.Icmp.Sge
  ) e1' e2' "tmp" builder
| SUnop(op, e) ->
  let e' = expr builder e in
  (match op with
    A.Neg      -> L.build_neg
  | A.Not      -> L.build_not)
  e' "tmp" builder
| SAssign (s, e) -> let e' = expr builder e in ignore(L.build_store e' (lookup
s) builder); e'
| SFuncall ("print", [e]) | SFuncall ("printb", [e]) ->
  L.build_call printf_func [| int_format_str ; (expr builder e) |]
  "printf" builder
| SFuncall ("printbig", [e]) ->
  L.build_call printbig_func [| (expr builder e) |] "printbig" builder
| SFuncall ("printf", [e]) ->
  L.build_call printf_func [| float_format_str ; (expr builder e) |]
  "printf" builder
| SFuncall (f, args) ->
  let (fdef, func_dec) = StringMap.find f function_decls in
  let llargs = List.rev (List.map (expr builder) (List.rev args)) in
  let result = (match func_dec.styp with
    A.Void -> ""

```

```

        | _ -> f ^ "_result") in
    L.build_call fdef (Array.of_list llargs) result builder
in

(* Invoke "f builder" if the current block doesn't already
   have a terminal (e.g., a branch). *)

let add_terminal builder f =
    match L.block_terminator (L.insertion_block builder) with
    | Some _ -> ()
    | None -> ignore (f builder)
in

(* generate code for different kinds of statements *)

let rec stmt builder = function
    SSeq s1 -> List.fold_left stmt builder s1
  | SExpr e -> ignore(expr builder e); builder
  | SVarDec ((typ, string), e) -> ignore(expr builder (typ, (SAssign(string, e))));
builder
  | SReturn e -> ignore (match func_dec.styp with
        (* Special "return nothing" instr *)
        A.Void -> L.build_ret_void builder
        (* Build return statement *)
        | _ -> L.build_ret (expr builder e) builder );
        builder
  | SIf (predicate, then_stmt, else_stmt) ->
    let bool_val = expr builder predicate in
    let merge_bb = L.append_block context "merge" the_function in
    let build_br_merge = L.build_br merge_bb in (* partial function *)

```

```

let then_bb = L.append_block context "then" the_function in
add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
  build_br_merge;

let else_bb = L.append_block context "else" the_function in
add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
  build_br_merge;

ignore (L.build_cond_br bool_val then_bb else_bb builder);
L.builder_at_end context merge_bb
| SWhile (predicate, body) ->
let pred_bb = L.append_block context "while" the_function in
ignore (L.build_br pred_bb builder);

let body_bb = L.append_block context "while_body" the_function in
add_terminal (stmt (L.builder_at_end context body_bb) body)
  (L.build_br pred_bb);

let pred_builder = L.builder_at_end context pred_bb in
let bool_val = expr pred_builder predicate in

let merge_bb = L.append_block context "merge" the_function in
ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
L.builder_at_end context merge_bb
in

(* Build the code for each statement in the function *)

```

```

let builder = stmt builder (SSeq func_dec.sbody) in

(* Add a return if the last block falls off the end *)
add_terminal builder (match func_dec.styp with
  | A.Void -> L.build_ret (L.const_float float_t 5.5)
  | A.Float -> L.build_ret (L.const_float float_t 0.0)
  | t -> L.build_ret (L.const_int (ltype_of_typ t) 5))
in

List.iter build_function_body functions;
the_module

```

Ast.ml

```

(* Abstract Syntax Tree and
functions for printing it *)

```

```

type op = Add | Sub | Mult | Mod | Div | Equal | Neq |
Less | Leq | Greater | Geq |
And | Or

```

```

type uop = Neg | Not

```

```

type typ = String | Int | Bool | Float | Void | Node of
typ

```

```

type bind = typ * string

```

```

type expr =
  Literal of int

```

```
| Fliteral of string
| BoolLit of bool
| Sliteral of string
| NodeLit of expr * expr
| NodeData of expr
| NodeParent of expr
| NodeLevel of expr
| LNodeShift of expr
| RNodeShift of expr
| SwapNode of expr * expr
| FindNode of expr * expr
| AddNode of expr * expr
| RemoveNode of expr * expr
| Id of string
| Binop of expr * op * expr
| Unop of uop * expr
| Assign of string * expr
| FunCall of string * expr list
| Noexpr
```

```
type stmt =
  (*Block of stmt list*)
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | VarDec of bind * expr
  (* | Assign of string * expr *)
  | For of expr * expr * expr * stmt
```

```

| While of expr * stmt
| Seq of stmt list

type func_decl = {
  typ: typ;
  fname: string;
  formals: bind list;
  body: stmt list;
}

type item =
  Stmt of stmt
  | Function of func_decl

type program = item list

(* Pretty-printing functions *)
let string_of_op = function
  Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Mod -> "%"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"

```



```
| Geq -> ">="
| And -> "&&"
| Or -> "||"
```

```
let string_of_uop = function
  Neg -> "-"
| Not -> "!"
```

```
let rec string_of_typ = function
  Int -> "int"
| Bool -> "bool"
| Float -> "float"
| Void -> "void"
| String -> "string"
| Node(t) -> "node<" ^ string_of_typ t ^ ">"
```

```
let rec string_of_expr = function
  Literal(l) -> string_of_int l
| Fliteral(l) -> l
| Sliteral(l) -> "\"" ^ l ^ "\""
| BoolLit(true) -> "true"
| BoolLit(false) -> "false"
| NodeLit(e1, e2) -> string_of_expr e1 ^ "," ^
string_of_expr e2
| NodeData(e) -> string_of_expr e ^ ".data"
| NodeParent(e) -> string_of_expr e ^ ".parent"
```

```

| NodeLevel(e) -> string_of_expr e ^ ".level"
| LNodeShift(e) -> string_of_expr e ^ "<<"
| RNodeShift(e) -> string_of_expr e ^ ">>"
| SwapNode(e1, e2) -> string_of_expr e1 ^ "^" ^
string_of_expr e2
| FindNode (g,param) -> string_of_expr g
    ^ ".find_node(" ^ string_of_expr param ^ ")"
| AddNode (g,param) -> string_of_expr g
    ^ ".add_node(" ^ string_of_expr param ^ ")"
| RemoveNode (g,param) -> string_of_expr g
    ^ ".remove_node(" ^ string_of_expr param ^ ")"
| Id(s) -> s
| Binop(e1, o, e2) -> string_of_expr e1 ^ " " ^
string_of_op o ^ " " ^ string_of_expr e2
| Unop(o, e) -> string_of_uop o ^ string_of_expr e
| FunCall(s,e) -> s ^ "(" ^ String.concat ", " (List.map
string_of_expr e) ^ ")"
| Assign(s,e) -> s ^ " = " ^ string_of_expr e
| Noexpr -> ""

```

```

let rec string_of_stmt = function

```

```

    Expr(expr) -> string_of_expr expr ^ ";\n";
| Return(expr) -> "return " ^ string_of_expr expr ^
";\n";
| If(e,s1,s2) -> "If(" ^ string_of_expr e ^ ") {" ^
string_of_stmt s1 ^ "} Else {" ^ string_of_stmt s2 ^ "}"
| VarDec((t, s),e) -> string_of_typ t ^ " " ^ s ^ " = " ^
string_of_expr e
(* | Assign(s,e) -> s ^ " = " ^ string_of_expr e *)
| For(e1, e2, e3, s) ->
    "for (" ^ string_of_expr e1 ^ " ; " ^ string_of_expr
e2 ^ " ; " ^
    string_of_expr e3 ^ ") " ^ string_of_stmt s

```

```

| While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^
string_of_stmt s
| Seq(l) -> String.concat ", " (List.map string_of_stmt
l)

```

```

let string_of_func_decl func_decl =
  func_decl.fname ^ "<" ^ string_of_typ func_decl.typ ^
">" ^ String.concat ", " (List.map snd
func_decl.formals) ^
"}{" ^
String.concat "" (List.map string_of_stmt func_decl.body)
^
"}\n"

```

```

let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^
";\n"

```

```

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
fdecl.fname ^ "(" ^ String.concat ", " (List.map snd
fdecl.formals) ^
")\n{\n" ^
String.concat "" (List.map string_of_vdecl fdecl.formals)
^
String.concat "" (List.map string_of_stmt fdecl.body) ^
"}\n"

```

```

let string_of_items item = match item with

```

```

    Stmt(x) -> string_of_stmt x
  |Function(y) -> string_of_fdecl y

let string_of_program stmts =
  String.concat " " (List.map string_of_items stmts)

```

Appendix: MicroC+

Appendix: MicroC+ C-Code:

Tree.h

```

#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include "tree.h"

// functions not defined in the header file
void del_node(struct Node *node);

void init_root(struct Node *node){
    node->root = 1;
}

struct Node *create_int_node(int data){
    struct Node *node = malloc(sizeof(struct Node));
    node->children = malloc(sizeof(struct List));
    node->data = malloc(sizeof(union data_u));

    node->children->head = 0;
    node->children->length = 0;

    node->level = 0;
    node->root = 0;
    node->next = NULL;
    node->prev = NULL;
    node->parent = NULL;

```

```

    node->dtype = INT;
    node->data->i = data;
    return node;
}

struct Node *create_char_node(char data){
    struct Node *node = malloc(sizeof(struct Node));
    node->children = malloc(sizeof(struct List));
    node->data = malloc(sizeof(union data_u));

    node->children->head = 0;
    node->children->length = 0;

    node->level = 0;
    node->root = 0;
    node->next = NULL;
    node->prev = NULL;
    node->parent = NULL;
    node->dtype = CHAR;
    node->data->c = data;
    return node;
}

struct Node *create_float_node(float data){
    struct Node *node = malloc(sizeof(struct Node));
    node->children = malloc(sizeof(struct List));
    node->data = malloc(sizeof(union data_u));

    node->children->head = 0;
    node->children->length = 0;

    node->level = 0;
    node->root = 0;
    node->next = NULL;
    node->prev = NULL;
    node->parent = NULL;
    node->dtype = FLOAT;
    node->data->f = data;
    return node;
}

void delete_node(struct Node *node){

```

```

    if(node->children->length != 0){ // if no children
        node->parent->children->length -= 1;
        del_node(node);
    }
    int i = 0;
    struct Node *curr = node->children->head;
    while(i < node->children->length){
        delete_node(curr);
        curr = curr->next;
        i++;
    }
}

void del_node(struct Node *node){
    free(node->children);
    free(node->data);
    free(node);
}

void add_child(struct Node *parent, struct Node *child){
    // TODO: check if child is ancestor of the parent already
    if(child->dtype != parent->dtype){
        perror("Cannot add a node of diffent data type to another
node");
        return;
    }
    if(child->root == 1){
        child->root = 0;
    }
    if(child->parent) // if was a child and head of another nod
        child->parent->children->head = child->next;
    child->parent = parent;
    child->level = parent->level + 1;
    if(parent->children->length == 0){
        parent->children->head = child;
        parent->children->length += 1;
    }
    int i = 0;
    struct Node *curr = parent->children->head;
    while(i < parent->children->length){
        if(curr->next)
            curr = curr->next;
    }
}

```

```

        i++;
    }
    curr->next = child;
    child->prev = curr;
    parent->children->length += 1;
}

struct Node *get_root(struct Node *n){ // could be done with lvl's
    if(n->root == 1) return n;

    struct Node *curr = n->parent;
    while(curr->root != 1 && curr->parent){
        if(curr->parent)
            curr = curr->parent;
    }
    if(curr->root == 1) return curr;
    return 0;
}

int is_empty(struct Node *node){
    return node->children->length == 0;
}

int is_ancestor(struct Node *parent, struct Node *child){
    struct Node *curr = child->parent;
    while(curr){
        if(curr == parent)
            return 1;
        curr = curr->parent;
    }
    return 0;
}

int is_root(struct Node *node){
    return node->root;
}

// int get_height(struct Node *n){
//     int height = 0;
//     int i = 0;
//     struct Node *curr = get_root(n);

```

```

// struct Node *curr = node->children->head;
// while(i < node->children->length){
//     print_tree(curr);
//     if(curr->next)
//         curr = curr->next;
//     i++;
// }
// return 0;
// }

int get_depth(struct Node *n){
    return n->level; // what if node is null
}

int get_index(struct Node *node){
    return 0;
}

// throw errors?
void shift_right(int index, struct Node *child){
    if(child == NULL) return;
    if(child->parent == NULL) return; // free floating node

    struct Node *ahead_place = child;
    int i = 0; // making the ahead place index times ahead of the other
    for(i = 0; i < index; i++){
        ahead_place = ahead_place->next;
    }
    if(!ahead_place){
        perror("Index out of range");
        return;
    }
    if(index == 1){
        struct Node *temp = ahead_place->next;
        ahead_place->next = child;
        child->next = temp;
        ahead_place->prev = child->prev;
        if(child->prev)
            child->prev->next = ahead_place;
        child->prev = ahead_place;
    }
}

```



```

else{
    struct Node *temp = ahead_place->next;
    if(ahead_place->next)
        ahead_place->next->prev = child;
    ahead_place->next = child->next;
    ahead_place->next->prev = ahead_place;
    child->next = temp;

    temp = ahead_place->prev;
    if(child->prev)
        ahead_place->prev->next = child->prev;
    ahead_place->prev = child->prev;
    child->prev = temp;
    child->prev->next = child;
}

if(child->parent->children->head == child){
    child->parent->children->head = ahead_place;
}

return;
}

void shift_left(int index, struct Node *child){
    if(child == NULL) return;
    if(child->parent == NULL) return; // free floating node

    struct Node *behind_place = child;
    int i = 0; // making the ahead place index times ahead of the other
    for(i = 0; i < index; i++){
        behind_place = behind_place->prev;
    }
    if(!behind_place){
        perror("Index out of range");
        return;
    }
    if(index == 1){
        shift_right(1, behind_place);
    }
    else{
        struct Node *temp = behind_place->next;
        if(child->next)

```

```

        child->next->prev = behind_place;
        behind_place->next->prev = child;
        behind_place->next = child->next;
        child->next = temp;

        temp = child->prev;
        if(behind_place->prev)
            behind_place->prev->next = child;
        child->prev = behind_place->prev;
        behind_place->prev = temp;
        behind_place->prev->next = behind_place;

        if(child->parent->children->head == behind_place){
            child->parent->children->head = child;
        }
    }
    return;
}

// BUG: in certain cases, when the children are from same parent,
// the head won't switch, even though the nodes will
void deep_swap(struct Node *node_a, struct Node *node_b){
    struct Node * temp = NULL;
    if(node_a->level == node_b->level){ // same level
        if(node_a->parent == node_b->parent){ // not same parent
            if(node_b->parent->children->head == node_b){// if node_a
is the head of child list
                node_b->parent->children->head = node_a;
            }
            else if(node_b->parent->children->head == node_a){// if
node_a is the head of child list
                node_b->parent->children->head = node_b;
            }
        }
        struct Node *curr = node_a->parent->children->head;
        while(curr){ // make prev -> next point to correct one
            if(curr->next == node_a){
                curr->next = node_b;
                break;
            }
            else if(curr->next == node_b){
                curr->next = node_a;
                break;
            }
        }
    }
}

```

```

        }
        curr = curr->next;
    }
}
else{
    node_a->prev->next = node_b;
    node_b->prev->next = node_a;
    temp = node_a->parent;
    node_a->parent = node_b->parent;
    node_b->parent = temp;
}

if(node_a->next == node_b){
    node_a->next = node_b->next;
    node_b->next = node_a;
}
else if(node_b->next == node_a){
    temp = node_a->next;
    node_a->next = node_b;
    node_b->next = temp;
}
else{
    temp = node_a->next;
    node_a->next = node_b->next;
    node_b->next = temp;
}

temp = node_a->prev;
node_a->prev = node_b->prev;
node_b->prev = temp;
}
return;
}

// make this a bfs printing
void print_tree(struct Node *node){
    print_node(node);
    int i = 1;
    struct Node *curr = node->children->head;
    while(i < node->children->length){
        print_tree(curr);
    }
}

```

```

        if(curr->next)
            curr = curr->next;
        i++;
    }
}

void print_node(struct Node *node){
    if(node == NULL) return;
    switch(node->dtype){
        case INT:
            printf("Level: %d\t", node->level);
            printf("Data: %d\n", node->data->i);
            break;
        case CHAR:
            printf("Level: %d\t", node->level);
            printf("Data: %c\n", node->data->c);
            break;
        case FLOAT:
            printf("Level: %d\t", node->level);
            printf("Data: %f\n", node->data->f);
            break;
        default:
            break;
    }
    return;
}

```

*** Tree.c***

```

#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include "tree.h"

// functions not defined in the header file
void del_node(struct Node *node);

```

```

void init_root(struct Node *node){
    node->root = 1;
}

struct Node *create_int_node(int data){
    struct Node *node = malloc(sizeof(struct Node));
    node->children = malloc(sizeof(struct List));
    node->data = malloc(sizeof(union data_u));

    node->children->head = 0;
    node->children->length = 0;

    node->level = 0;
    node->root = 0;
    node->next = NULL;
    node->prev = NULL;
    node->parent = NULL;
    node->dtype = INT;
    node->data->i = data;
    return node;
}

struct Node *create_char_node(char data){
    struct Node *node = malloc(sizeof(struct Node));
    node->children = malloc(sizeof(struct List));
    node->data = malloc(sizeof(union data_u));

    node->children->head = 0;
    node->children->length = 0;

    node->level = 0;
    node->root = 0;
    node->next = NULL;
    node->prev = NULL;
    node->parent = NULL;
    node->dtype = CHAR;
    node->data->c = data;
    return node;
}

struct Node *create_float_node(float data){

```

```

struct Node *node = malloc(sizeof(struct Node));
node->children = malloc(sizeof(struct List));
node->data = malloc(sizeof(union data_u));

node->children->head = 0;
node->children->length = 0;

node->level = 0;
node->root = 0;
node->next = NULL;
node->prev = NULL;
node->parent = NULL;
node->dtype = FLOAT;
node->data->f = data;
return node;
}

void delete_node(struct Node *node){
    if(node->children->length != 0){ // if no children
        node->parent->children->length -= 1;
        del_node(node);
    }
    int i = 0;
    struct Node *curr = node->children->head;
    while(i < node->children->length){
        delete_node(curr);
        curr = curr->next;
        i++;
    }
}

void del_node(struct Node *node){
    free(node->children);
    free(node->data);
    free(node);
}

void add_child(struct Node *parent, struct Node *child){
    // TODO: check if child is ancestor of the parent already
    if(child->dtype != parent->dtype){
        perror("Cannot add a node of different data type to another
node");
    }
}

```

```

        return;
    }
    if(child->root == 1){
        child->root = 0;
    }
    if(child->parent) // if was a child and head of another node
        child->parent->children->head = child->next;
    child->parent = parent;
    child->level = parent->level + 1;
    if(parent->children->length == 0){
        parent->children->head = child;
        parent->children->length += 1;
    }
    int i = 0;
    struct Node *curr = parent->children->head;
    while(i < parent->children->length){
        if(curr->next)
            curr = curr->next;
        i++;
    }
    curr->next = child;
    child->prev = curr;
    parent->children->length += 1;
}

struct Node *get_root(struct Node *n){ // could be done with lvl's
    if(n->root == 1) return n;

    struct Node *curr = n->parent;
    while(curr->root != 1 && curr->parent){
        if(curr->parent)
            curr = curr->parent;
    }
    if(curr->root == 1) return curr;
    return 0;
}

int is_empty(struct Node *node){
    return node->children->length == 0;
}

```

```

int is_ancestor(struct Node *parent, struct Node *child){
    struct Node *curr = child->parent;
    while(curr){
        if(curr == parent)
            return 1;
        curr = curr->parent;
    }
    return 0;
}

int is_root(struct Node *node){
    return node->root;
}

// int get_height(struct Node *n){
//     int height = 0;
//     int i = 0;
//     struct Node *curr = get_root(n);
//     struct Node *curr = node->children->head;
//     while(i < node->children->length){
//         print_tree(curr);
//         if(curr->next)
//             curr = curr->next;
//         i++;
//     }
//     return 0;
// }

int get_depth(struct Node *n){
    return n->level; // what if node is null
}

int get_index(struct Node *node){
    return 0;
}

// throw errors?
void shift_right(int index, struct Node *child){
    if(child == NULL) return;
    if(child->parent == NULL) return; // free floating node
}

```



```

struct Node *ahead_place = child;
int i = 0; // making the ahead place index times ahead of the other
for(i = 0; i < index; i++){
    ahead_place = ahead_place->next;
}
if(!ahead_place){
    perror("Index out of range");
    return;
}
if(index == 1){
    struct Node *temp = ahead_place->next;
    ahead_place->next = child;
    child->next = temp;
    ahead_place->prev = child->prev;
    if(child->prev)
        child->prev->next = ahead_place;
    child->prev = ahead_place;
}
else{
    struct Node *temp = ahead_place->next;
    if(ahead_place->next)
        ahead_place->next->prev = child;
    ahead_place->next = child->next;
    ahead_place->next->prev = ahead_place;
    child->next = temp;

    temp = ahead_place->prev;
    if(child->prev)
        ahead_place->prev->next = child->prev;
    ahead_place->prev = child->prev;
    child->prev = temp;
    child->prev->next = child;
}

if(child->parent->children->head == child){
    child->parent->children->head = ahead_place;
}

return;
}

void shift_left(int index, struct Node *child){

```

```

        if(child == NULL) return;
    if(child->parent == NULL) return; // free floating node

    struct Node *behind_place = child;
    int i = 0; // making the ahead place index times ahead of the other
    for(i = 0; i < index; i++){
        behind_place = behind_place->prev;
    }
    if(!behind_place){
        perror("Index out of range");
        return;
    }
    if(index == 1){
        shift_right(1, behind_place);
    }
    else{
        struct Node *temp = behind_place->next;
        if(child->next)
            child->next->prev = behind_place;
        behind_place->next->prev = child;
        behind_place->next = child->next;
        child->next = temp;

        temp = child->prev;
        if(behind_place->prev)
            behind_place->prev->next = child;
        child->prev = behind_place->prev;
        behind_place->prev = temp;
        behind_place->prev->next = behind_place;

        if(child->parent->children->head == behind_place){
            child->parent->children->head = child;
        }
    }
    return;
}

// BUG: in certain cases, when the children are from same parent,
// the head won't switch, even though the nodes will
void deep_swap(struct Node *node_a, struct Node *node_b){
    struct Node * temp = NULL;
    if(node_a->level == node_b->level){ // same level

```

```

        if(node_a->parent == node_b->parent){ // not same parent
            if(node_b->parent->children->head == node_b){// if node_a
is the head of child list
                node_b->parent->children->head = node_a;
            }
            else if(node_b->parent->children->head == node_a){// if
node_a is the head of child list
                node_b->parent->children->head = node_b;
            }
            struct Node *curr = node_a->parent->children->head;
            while(curr){ // make prev -> next point to correct one
                if(curr->next == node_a){
                    curr->next = node_b;
                    break;
                }
                else if(curr->next == node_b){
                    curr->next = node_a;
                    break;
                }
                curr = curr->next;
            }
        }
    }
    else{
        node_a->prev->next = node_b;
        node_b->prev->next = node_a;
        temp = node_a->parent;
        node_a->parent = node_b->parent;
        node_b->parent = temp;
    }

    if(node_a->next == node_b){
        node_a->next = node_b->next;
        node_b->next = node_a;
    }
    else if(node_b->next == node_a){
        temp = node_a->next;
        node_a->next = node_b;
        node_b->next = temp;
    }
    else{
        temp = node_a->next;

```

```

        node_a->next = node_b->next;
        node_b->next = temp;
    }

    temp = node_a->prev;
    node_a->prev = node_b->prev;
    node_b->prev = temp;
}
return;
}

// make this a bfs printing
void print_tree(struct Node *node){
    print_node(node);
    int i = 1;
    struct Node *curr = node->children->head;
    while(i < node->children->length){
        print_tree(curr);
        if(curr->next)
            curr = curr->next;
        i++;
    }
}

void print_node(struct Node *node){
    if(node == NULL) return;
    switch(node->dtype){
        case INT:
            printf("Level: %d\t", node->level);
            printf("Data: %d\n", node->data->i);
            break;
        case CHAR:
            printf("Level: %d\t", node->level);
            printf("Data: %c\n", node->data->c);
            break;
        case FLOAT:
            printf("Level: %d\t", node->level);
            printf("Data: %f\n", node->data->f);
            break;
        default:
            break;
    }
}

```

```
    return;  
}
```

Treetest.c

```
// DELETE AFTER DONE TESTING  
#include <stdio.h>  
#include "tree.h"  
  
// comment this out for valgrind testing  
void test_create_int_node(){  
    int p = 5;  
    void *q = &p;  
    struct Node * n = create_node(q, INT);  
    print_node(n);  
  
    int a = 11;  
    void *b = &a;  
    struct Node *nn = create_node(b, INT);  
    print_node(nn);  
  
    delete_node(n);  
    delete_node(nn);  
}  
  
void test_delete_node(){  
    int p = 77;  
    void *q = &p;  
    struct Node * n = create_node(q, INT);  
  
    delete_node(n);  
}  
  
void test_init_root(){  
    int p = 5;  
    void *q = &p;  
    struct Node * n = create_node(q, INT);  
    printf("is_root 0 == %d\n", n->root);  
    init_root(n);  
    printf("is_root 1 == %d\n", n->root);  
}
```

```

    delete_node(n);
}

void test_add_child(){
    int p = 5; // root
    void *q = &p;
    struct Node * parent = create_node(q, INT);

    int r = 77; // root child
    void *s = &r;
    struct Node * nchild = create_node(s, INT);

    int t = 71; // nchild's child
    void *u = &t;
    struct Node * nnchild = create_node(u, INT);

    int a = 11; // root child
    void *b = &a;
    struct Node *child = create_node(b, INT);
    add_child(parent, child);

    int c = 21; // root child
    void *d= &c;
    struct Node *nnnchild = create_node(d, INT);

    add_child(parent, nchild);
    add_child(parent, nnnchild);
    add_child(nchild, nnchild);
    print_tree(parent);
}

void test_deep_swap_same_level(){
    int p = 5; // root
    void *q = &p;
    struct Node * parent = create_node(q, INT);

    int r = 77; // root child
    void *s = &r;
    struct Node * nchild = create_node(s, INT);

    int a = 11; // root child
    void *b = &a;

```

```

    struct Node *child = create_node(b, INT);

    // int c = 21; // root child
    // void *d= &c;
    // struct Node *nchild = create_node(d, INT);

    add_child(parent, child);
    add_child(parent, nchild);
    // add_child(parent, nchild);

    deep_swap(child, nchild);
    print_tree(parent);
    printf("\n");
    deep_swap(nchild, child);
    print_tree(parent);

    printf("\n");
    deep_swap(child, nchild);
    print_tree(parent);
}

void test_shift_right_one(){
    int p = 5; // root
    void *q = &p;
    struct Node * parent = create_node(q, INT);

    int r = 77; // root child
    void *s = &r;
    struct Node * nchild = create_node(s, INT);

    int a = 11; // root child
    void *b = &a;
    struct Node *child = create_node(b, INT);

    add_child(parent, child);
    add_child(parent, nchild);
    print_tree(parent);
    printf("\n");
    shift_right(1, child);
    print_tree(parent);
}

```

```

int main(){
    printf("\n-----create_node-----\n");
    test_create_int_node();

    printf("\n-----delete_node-----\n");
    // test_delete_node();

    printf("\n-----init_root-----\n");
    // test_init_root();

    printf("\n-----add_child-----\n");
    test_add_child();

    printf("\n-----deep_swap-----\n");
    // test_deep_swap_same_level();

    printf("\n-----shift_right-----\n");
    // test_shift_right_one();

    // // Test init root
    // printf("\n-----init_root-----\n");
    // int p = 5;
    // void * q = &p;
    // struct Node *empty_root = create_node(q);
    // init_root(empty_root);
    // print_node(empty_root, &print_int);

    // int t = 1;
    // void * s = &t;
    // struct Node *tree_root = create_node(s);
    // init_root(tree_root);
    // print_node(tree_root, &print_int);

    // // Test create_node
    // printf("\n-----create_node-----\n");
    // int a = 77;
    // void * b = &a;
    // struct Node *n = create_node(b);
    // print_node(n, &print_int);

    // int c = 121;
    // void * d = &c;

```



```

// struct Node *nn = create_node(d);
// print_node(nn, &print_int);

// int cc = 11;
// void * dd = &cc;
// struct Node *nnn = create_node(dd);
// print_node(nnn, &print_int);

// int ccc = 27;
// void * ddd = &ccc;
// struct Node *m = create_node(ddd);
// print_node(m, &print_int);

// test add_child
// printf("\n-----add_child-----\n");
// add_child(tree_root, nn);
// print_tree(tree_root, &print_int);
// printf("\n");

// add_child(tree_root, nnn);
// print_tree(tree_root, &print_int);
// printf("\n");

// add_child(nn, m);
// print_tree(tree_root, &print_int);

// // test is_empty
// printf("\n-----is_empty-----\n");
// // printf("is_empty 1 == %d\n", is_empty(empty_root)); // tree is
not empty
// // printf("is_empty 0 == %d\n", is_empty(tree_root)); // tree is
not empty
// // need to test if the tree is empty after adding a child to the
tree

// //test is root
// printf("\n-----is_root-----\n");
// printf("is_root 1 == %d\n", is_root(empty_root)); // tree is root
// printf("is_root 1 == %d\n", is_root(tree_root)); // tree is root
// printf("is_root 0 == %d\n", is_root(n)); // tree is not root and
not part of a tree
// // printf("is_root 0 == %d\n", is_root(n)); // tree is not root

```

```

part of a tree
    // // need to find the root for a child in the tree

    // printf("\n-----get_depth-----\n");
    // printf("get_depth: 1 == %d\n", get_depth(empty_root)); // is a
root
    // printf("get_depth: 1 == %d\n", get_depth(tree_root)); // is a root
    // printf("get_depth: 2 == %d\n", get_depth(nn)); // is a child of a
tree
    // printf("get_depth: 0 == %d\n", get_depth(n)); // n is a node not
part of a tree
    // // Need to find the height of a child in the tree for tree with
multiple children

    // printf("\n-----get_height-----\n");
    // printf("get_height: 1 == %d\n", get_height(empty_root)); // tree
is a root
    // printf("get_height: 2 == %d\n", get_height(tree_root)); // tree is
a root
    // printf("get_height: 2 == %d\n", get_height(nn)); // n is a node
not part of a tree
    // printf("get_height: 0 == %d\n", get_height(n)); // n is a node not
part of a tree
    // // Need to find the height of a child in the tree for tree with
multiple children

    // printf("\n-----get_root-----\n");
    // print_node(get_root(empty_root), &print_int); // tree is a root
    // print_node(get_root(n), &print_int); // tree is not a root and not
part of a tree
    // // need ot fnd the root for a child in the tree
    // print_node(get_root(tree_root), &print_int);

    // printf("\n-----is_tree-----\n");
    // printf("Is_tree: 1 == %d\n", is_tree(empty_root));
    // printf("Is_tree: 1 == %d\n", is_tree(tree_root));
    // printf("Is_tree: 0 == %d\n", is_tree(n));

    // printf("\n-----print_tree-----\n");

```

```

// print_tree(empty_root, &print_int);
// printf("\n");
// print_tree(tree_root, &print_int);
// printf("\n");
// print_tree(n, &print_int);

// printf("\n-----print_node-----\n");
// print_node(empty_root, &print_int);
// print_node(tree_root, &print_int);
// print_node(n, &print_int);

// printf("\n-----is_ancestor-----\n");
// printf("is_ancestor 0 == %d\n", is_ancestor(tree_root, n)); //
unrelated node
// printf("is_ancestor 0 == %d\n", is_ancestor(empty_root, nn));
// printf("is_ancestor 1 == %d\n", is_ancestor(tree_root, nn));
// // need to test for multiple children to a parent

// printf("\n-----get_index-----\n");
// printf("get_index 0 == %d\n", get_index(tree_root));
// printf("get_index 0 == %d\n", get_index(empty_root));
// printf("get_index 1 == %d\n", get_index(nn));
// printf("get_index 0 == %d\n", get_index(n)); // unrelated node

// printf("\n-----shift_right-----\n");
// shift_right(0, tree_root);
// print_tree(tree_root, &print_int);
// shift_right(0, empty_root);
// print_tree(empty_root, &print_int);

// printf("\n-----deep_swap-----\n");
// deep_swap(tree_root, nn);
// print_tree(tree_root, &print_int);

return 0;
}

```

Makefile

```

CCCC = gcc
INCLUDES =

```

```

CFLAGS = -g -Wall $(INCLUDES)
LDFLAGS = -g

MAIN = tree

$(MAIN): treetest.o tree.o

.PHONEY: clean
clean:
    rm -f *.o $(MAIN)

.PHONEY: run
run: $(MAIN)
    ./$(MAIN)

.PHONEY: valgrind
valgrind: $(MAIN)
    valgrind --leak-check=full ./$(MAIN)

```

Appendix: MicroC+ Code:

Scanner

```

(* Ocamllex scanner for MicroC *)

{ open Microcparse
}

let digit = ['0' - '9']
let digits = digit+

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/" * { comment lexbuf } (* Comments *)
| '(' { LPAREN }
| ')' { RPAREN }
| '{' { LBRACE }
| '}' { RBRACE }
| '[' { LBRACK }

```

```

| ']'      { RBRACK  }
| ';'     { SEMI   }
| ','     { COMMA  }
| '+'     { PLUS   }
| '-'     { MINUS  }
| '*'     { TIMES  }
| '%'     { MOD    }
| '/'     { DIVIDE }
| '='     { ASSIGN }
| "node"  { NODE   }
| ".data" { DATA  }
| ".parent" { PARENT }
| ".level" { NODE_LEVEL }
| "<<"    { LSHIFT_NODE }
| ">>"    { RSHIFT_NODE }
| "^"     { SWAP_NODE }
| ".find_node" { FIND_NODE }
| ".add_node" { ADD_NODE }
| ".remove_node" { REMOVE_NODE }
| "=="    { EQ }
| "!="    { NEQ }
| '<'    { LT }
| "<="   { LEQ }
| '>'    { GT }
| ">="   { GEQ }
| "&&"   { AND }
| "||"   { OR }
| "!"    { NOT }
| "if"   { IF }
| "else" { ELSE }
| "for"  { FOR }
| "while" { WHILE }
| "return" { RETURN }
| "string" { STRING }
| "int"   { INT }
| "bool"  { BOOL }
| "float" { FLOAT }
| "void"  { VOID }
| "true"  { BLIT(true) }
| "false" { BLIT(false) }
| '\\'' ([^\\\'"]* as lxm) '\\'' { SLITERAL(lxm) }
| digits as lxm { LITERAL(int_of_string lxm) }

```

```

| digits '.' digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_' ]* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "*/" { token lexbuf }
| _ { comment lexbuf }

```

Parser

```

/* Ocaml yacc parser for MicroC */

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA PLUS MINUS
TIMES MOD DIVIDE ASSIGN
%token NOT EQ NEQ LT LEQ GT GEQ AND OR
%token RETURN IF ELSE FOR WHILE STRING INT BOOL FLOAT VOID
%token <int> LITERAL
%token <string> SLITERAL
%token <bool> BLIT
%token <string> ID FLIT
%token EOF
%token DATA NODE_LEVEL NODE PARENT LSHIFT_NODE RSHIFT_NODE SWAP_NODE
ADD_NODE REMOVE_NODE FIND_NODE

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS

```

```

%left TIMES DIVIDE MOD
%right NOT
%left PARENT
%left DATA NODE_LEVEL LSHIFT_NODE RSHIFT_NODE ADD_NODE REMOVE_NODE
SWAP_NODE FIND_NODE

%%

program:
  decls EOF { $1 }

decls:
  /* nothing */ { [], [] }
  | decls vdecl { (($2 :: fst $1), snd $1) }
  | decls fdecl { (fst $1, ($2 :: snd $1)) }

fdecl:
  typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
  { { typ = $1;
    fname = $2;
    formals = List.rev $4;
    locals = List.rev $7;
    body = List.rev $8 } }

formals_opt:
  /* nothing */ { [] }
  | formal_list { $1 }

formal_list:
  typ ID { [($1,$2)] }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }

typ:
  INT { Int }
  | STRING { String }
  | BOOL { Bool }
  | FLOAT { Float }
  | VOID { Void }
  | NODE LT typ GT { Node($3) }

vdecl_list:
  /* nothing */ { [] }

```

```

| vdecl_list vdecl { $2 :: $1 }

vdecl:
  typ ID SEMI { ($1, $2) }

stmt_list:
  /* nothing */ { [] }
| stmt_list stmt { $2 :: $1 }

stmt:
  expr SEMI { Expr $1 }
| RETURN expr_opt SEMI { Return $2 }
| LBRACE stmt_list RBRACE { Block(List.rev $2) }
| IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
| IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
| FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
  { For($3, $5, $7, $9) }
| WHILE LPAREN expr RPAREN stmt { While($3, $5) }

expr_opt:
  /* nothing */ { Noexpr }
| expr { $1 }

expr:
  LITERAL { Literal($1) }
| SLITERAL { Sliteral($1) }
| FLIT { Fliteral($1) }
| BLIT { BoolLit($1) }
| ID { Id($1) }
| expr PLUS expr { Binop($1, Add, $3) }
| expr MINUS expr { Binop($1, Sub, $3) }
| expr TIMES expr { Binop($1, Mult, $3) }
| expr MOD expr { Binop($1, Mod, $3) }
| expr DIVIDE expr { Binop($1, Div, $3) }
| expr EQ expr { Binop($1, Equal, $3) }
| expr NEQ expr { Binop($1, Neq, $3) }
| expr LT expr { Binop($1, Less, $3) }
| expr LEQ expr { Binop($1, Leq, $3) }
| expr GT expr { Binop($1, Greater, $3) }
| expr GEQ expr { Binop($1, Geq, $3) }
| expr AND expr { Binop($1, And, $3) }
| expr OR expr { Binop($1, Or, $3) }

```



```

| MINUS expr %prec NOT { Unop(Neg, $2)      }
| NOT expr           { Unop(Not, $2)        }
| ID ASSIGN expr    { Assign($1, $3)       }
| ID LPAREN args_opt RPAREN { Call($1, $3)  }
| LPAREN expr RPAREN { $2                  }

/* Node Auxillary*/

| expr DATA { NodeData($1) }
/* | expr DATA ASSIGN expr { Assign(NodeData($1), $4)}
| expr PARENT ASSIGN expr { Assign(NodeParent($1), $4) }*/
| expr PARENT { NodeParent($1) }
| LBRACK expr COMMA expr RBRACK { NodeLit($2, $4) }
| expr NODE_LEVEL { NodeLevel($1) }
| expr LSHIFT_NODE { LNodeShift($1)}
| expr RSHIFT_NODE { RNodeShift($1)}
| expr SWAP_NODE expr { SwapNode($1, $3) }
| expr FIND_NODE LPAREN expr RPAREN { FindNode($1, $4) }
| expr ADD_NODE LPAREN expr RPAREN { AddNode($1, $4) }
| expr REMOVE_NODE LPAREN expr RPAREN { RemoveNode($1, $4) }

args_opt:
  /* nothing */ { [] }
| args_list { List.rev $1 }

args_list:
  expr { [$1] }
| args_list COMMA expr { $3 :: $1 }

```

Ast

```

(* Abstract Syntax Tree and functions for printing it *)

type op = Add | Sub | Mult | Mod | Div | Equal | Neq | Less | Leq | Greater
| Geq |
  And | Or

type uop = Neg | Not

type typ = String | Int | Bool | Float | Void | Node of typ

```

```

type bind = typ * string

type expr =
  Literal of int
  | Fliteral of string
  | BoolLit of bool
  | Sliteral of string
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr
  | NodeLit of expr * expr
  | NodeData of expr
  | NodeParent of expr
  | NodeLevel of expr
  | LNodeShift of expr
  | RNodeShift of expr
  | SwapNode of expr * expr
  | FindNode of expr * expr
  | AddNode of expr * expr
  | RemoveNode of expr * expr

type stmt =
  Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

type func_decl = {
  typ : typ;
  fname : string;
  formals : bind list;
  locals : bind list;
  body : stmt list;
}

type program = bind list * func_decl list

```

```

(* Pretty-printing functions *)

let string_of_op = function
  Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Mod -> "%"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let string_of_uop = function
  Neg -> "-"
  | Not -> "!"

let rec string_of_expr = function
  Literal(l) -> string_of_int l
  | Sliteral(l) -> "\"" ^ l ^ "\""
  | Fliteral(l) -> l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | Id(s) -> s
  | Binop(e1, o, e2) ->
    string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, e1) ->
    f ^ "(" ^ String.concat ", " (List.map string_of_expr e1) ^ ")"
  | Noexpr -> ""
  | NodeLit(e1, e2) -> string_of_expr e1 ^ "," ^ string_of_expr e2
  | NodeData(e) -> string_of_expr e ^ ".data"
  | NodeParent(e) -> string_of_expr e ^ ".parent"
  | NodeLevel(e) -> string_of_expr e ^ ".level"
  | LNodeShift(e) -> string_of_expr e ^ "<<"
  | RNodeShift(e) -> string_of_expr e ^ ">>"

```

```

| SwapNode(e1, e2) -> string_of_expr e1 ^ "^" ^ string_of_expr e2
| FindNode (g,param) -> string_of_expr g
    ^ ".find_node(" ^ string_of_expr param ^ ")"
| AddNode (g,param) -> string_of_expr g
    ^ ".add_node(" ^ string_of_expr param ^ ")"
| RemoveNode (g,param) -> string_of_expr g
    ^ ".remove_node(" ^ string_of_expr param ^ ")"

let rec string_of_stmt = function
  Block(stmts) ->
    "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s
  | If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ")\n" ^
    string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
    "for (" ^ string_of_expr e1 ^ " ; " ^ string_of_expr e2 ^ " ; " ^
    string_of_expr e3 ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let rec string_of_typ = function
  Int -> "int"
  | String -> "string"
  | Bool -> "bool"
  | Float -> "float"
  | Void -> "void"
  | Node(t) -> "node<" ^ string_of_typ t ^ ">"

let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^

```

```
String.concat "\n" (List.map string_of_fdecl funcs)
```

Sast

```
(* Semantically-checked Abstract Syntax Tree and functions for printing it *)
```

```
open Ast
```

```
type sexpr = typ * sx
```

```
and sx =
```

```
  SLiteral of int
| SFliteral of string
| SBoolLit of bool
| SSliteral of string
| SId of string
| SBinop of sexpr * op * sexpr
| SUnop of uop * sexpr
| SAssign of string * sexpr
| SCall of string * sexpr list
| SNoexpr
| SNodeLit of sexpr * sexpr
| SNodeData of sexpr
| SNodeParent of sexpr
| SNodeLevel of sexpr
| SLNodeShift of sexpr
| SRNodeShift of sexpr
| SSwapNode of sexpr * sexpr
| SFindNode of sexpr * sexpr
| SAddNode of sexpr * sexpr
| SRemoveNode of sexpr * sexpr
```

```
type sstmt =
```

```
  SBlock of sstmt list
| SExpr of sexpr
| SReturn of sexpr
| SIf of sexpr * sstmt * sstmt
| SFor of sexpr * sexpr * sexpr * sstmt
| SWhile of sexpr * sstmt
```

```

type sfunc_decl = {
  styp : typ;
  sfname : string;
  sformals : bind list;
  slocals : bind list;
  sbody : sstmt list;
}

type sprogram = bind list * sfunc_decl list

(* Pretty-printing functions *)

let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (match e with
    | SLiteral(l) -> string_of_int l
    | SSLiteral(l) -> "\"" ^ l ^ "\""
    | SBoolLit(true) -> "true"
    | SBoolLit(false) -> "false"
    | SFliteral(l) -> l
    | SId(s) -> s
    | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
    | SUNop(o, e) -> string_of_uop o ^ string_of_sexpr e
    | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
    | SCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
    | SNodeLit(e1, e2) -> string_of_sexpr e1 ^ "," ^ string_of_sexpr e2
    | SNodeData(e) -> string_of_sexpr e ^ ".data"
    | SNodeParent(e) -> string_of_sexpr e ^ ".parent"
    | SNodeLevel(e) -> string_of_sexpr e ^ ".level"
    | SLNodeShift(e) -> string_of_sexpr e ^ "<<"
    | SRNodeShift(e) -> string_of_sexpr e ^ ">>"
    | SSwapNode(e1, e2) -> string_of_sexpr e1 ^ "^" ^ string_of_sexpr e2
    | SFindNode (g,param) -> string_of_sexpr g
      ^ ".find_node(" ^ string_of_sexpr param ^ ")"
    | SAddNode (g,param) -> string_of_sexpr g
      ^ ".add_node(" ^ string_of_sexpr param ^ ")"
    | SRemoveNode (g,param) -> string_of_sexpr g
      ^ ".remove_node(" ^ string_of_sexpr param ^ ")"
    | SNoexpr -> ""
  ) ^ ")"

```

```

let rec string_of_sstmt = function
  SBlock(stmts) ->
    "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
  | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
  | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
  | SIf(e, s, SBlock([])) ->
    "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
  | SIf(e, s1, s2) -> "if (" ^ string_of_sexpr e ^ ")\n" ^
    string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SFor(e1, e2, e3, s) ->
    "for (" ^ string_of_sexpr e1 ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
    string_of_sexpr e3 ^ ") " ^ string_of_sstmt s
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt
s

let string_of_sfdecl fdecl =
  string_of_typ fdecl.styp ^ " " ^
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sformals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.slocals) ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "}\n"

let string_of_sprogram (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_sfdecl funcs)

```

Codegen

```
(* Code generation: translate takes a semantically checked AST and produces LLVM IR
```

LLVM tutorial: Make sure to read the OCaml version of the tutorial

<http://llvm.org/docs/tutorial/index.html>

Detailed documentation on the OCaml LLVM library:

<http://llvm.moe/>

<http://llvm.moe/ocaml/>

```

newverison
*)

module L = Llvml
module A = Ast
open Sast

module StringMap = Map.Make(String)

(* translate : Sast.program -> Llvml.module *)
let translate (globals, functions) =
  let context = L.global_context () in

  (* Create the LLVM compilation module into which
     we will generate code *)
  (* let llmem = L.MemoryBuffer.of_file "tree.bc" in
     let llm = Llvml_bitreader.parse_bitcode context llmem in *)
  let the_module = L.create_module context "MicroC" in

  (* Get types from the context *)
  let i32_t = L.i32_type context
  and i8_t = L.i8_type context
  and i1_t = L.i1_type context
  and str_t = L.pointer_type (L.i8_type context)
  and float_t = L.double_type context
  and void_ptr_t = L.pointer_type (L.i8_type context)
  and void_t = L.void_type context
  (* and node_t = L.pointer_type (match L.type_by_name llm "struct.Node"
with
  None -> raise (Failure "Missing implementation for struct Node")
  | Some t -> t) in *)
  in
  (* Return the LLVM type for a MicroC type *)
  let ltype_of_typ = function
    A.Int -> i32_t
  (* | A.Char -> i8_t *)
  | A.String -> str_t
  | A.Bool -> i1_t
  | A.Float -> float_t
  | A.Void -> void_t
  (* | A.Node(_) -> node_t *)
  in

```



```

(*)
  (* node functions *)
  let create_node_t = L.function_type node_t [| void_ptr_t ; i32_t |] in
  let create_node_f = L.declare_function "create_node" create_node_t
the_module in
  let root_init_t = L.function_type node_t [| node_t |] in
  let root_init_f = L.declare_function "root_init" root_init_t the_module
in
  let delete_node_t = L.function_type void_ptr_t [| node_t |] in
  let delete_node_f = L.declare_function "delete_node" delete_node_t
the_module in
  let add_child_t = L.function_type node_t [| node_t; node_t|] in
  let add_child_f = L.declare_function "add_child" add_child_t the_module
in
  let get_root_t = L.function_type node_t [| node_t |] in
  let get_root_f = L.declare_function "get_root" get_root_t the_module in
  let is_ancestor_t = L.function_type i32_t [| node_t ; node_t |] in
  let is_ancestor_f = L.declare_function "is_ancestor" is_ancestor_t
the_module in
  let is_root_t = L.function_type i32_t [| node_t |] in
  let is_root_f = L.declare_function "is_root" is_root_t the_module in
  let get_depth_t = L.function_type i32_t [| node_t |] in
  let get_depth_f = L.declare_function "get_depth" get_depth_t the_module
in
  let shift_right_t = L.function_type void_t [| i32_t; node_t |] in
  let shift_right_f = L.declare_function "shift_right" shift_right_t
the_module in
  let shift_left_t = L.function_type void_t [| i32_t; node_t |] in
  let shift_left_f = L.declare_function "shift_left" shift_left_t
the_module in
*)
(* Create a map of global variables after creating each *)
let global_vars : L.llvalue StringMap.t =
  let global_var m (t, n) =
    let init = match t with
      A.Float -> L.const_float (ltype_of_typ t) 0.0
    | _ -> L.const_int (ltype_of_typ t) 0
    in StringMap.add n (L.define_global n init the_module) m in
  List.fold_left global_var StringMap.empty globals in

let printf_t : L.lltype =

```

```

    L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let printf_func : L.llvalue =
    L.declare_function "printf" printf_t the_module in

let create_int_node : L.lltype =
    L.var_arg_function_type i32_t [| i32_t |] in
let create_int_node_func : L.llvalue =
    L.declare_function "create_int_node" create_int_node the_module
in
let amalg : L.lltype =
    L.function_type i32_t [| i32_t |] in
let amalg_func : L.llvalue =
    L.declare_function "amalg" amalg the_module in

let printbig_t : L.lltype =
    L.function_type i32_t [| i32_t |] in
let printbig_func : L.llvalue =
    L.declare_function "printbig" printbig_t the_module in

(* Define each function (arguments and return type) so we can
   call it even before we've created its body *)
let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
    let function_decl m fdecl =
        let name = fdecl.sfname
        and formal_types =
            Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.sformals)
        in let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types
    in
    StringMap.add name (L.define_function name ftype the_module, fdecl) m
in
    List.fold_left function_decl StringMap.empty functions in

(* Fill in the body of the given function *)
let build_function_body fdecl =
    let (the_function, _) = StringMap.find fdecl.sfname function_decls in
    let builder = L.builder_at_end context (L.entry_block the_function) in

    let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
    and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder
    and string_format_str = L.build_global_stringptr "%s\n" "fmt" builder
in
    (* and char_format_str = L.build_global_stringptr "%c\n" "fmt" builder

```

```

in *)

(* Construct the function's "locals": formal arguments and locally
   declared variables. Allocate each on the stack, initialize their
   value, if appropriate, and remember their values in the "locals" map
*)
let local_vars =
  let add_formal m (t, n) p =
    L.set_value_name n p;
  let local = L.build_alloca (ltype_of_typ t) n builder in
  ignore (L.build_store p local builder);
  StringMap.add n local m

  (* Allocate space for any locally declared variables and add the
   * resulting registers to our map *)
  and add_local m (t, n) =
  let local_var = L.build_alloca (ltype_of_typ t) n builder
  in StringMap.add n local_var m
  in

  let formals = List.fold_left2 add_formal StringMap.empty
fdecl.sformals
  (Array.to_list (L.params the_function)) in
  List.fold_left add_local formals fdecl.slocals
in

(* Return the value for a variable or formal argument.
   Check local names first, then global names *)
let lookup n = try StringMap.find n local_vars
  with Not_found -> StringMap.find n global_vars
in

(* Construct code for an expression; return its value *)
let rec expr builder ((_, e) : sexpr) = match e with
  SLiteral i -> L.const_int i32_t i
  | SSLiteral s -> L.build_global_stringptr s "str" builder
  (* | SCLiteral c -> L.build_global_stringptr c "str" builder *)
  | SBoolLit b -> L.const_int i1_t (if b then 1 else 0)
  | SFliteral l -> L.const_float_of_string float_t l
  | SNoexpr -> L.const_int i32_t 0
  | SId s -> L.build_load (lookup s) s builder
  | SAssign (s, e) -> let e' = expr builder e in

```

```

        ignore(L.build_store e' (lookup s) builder); e'
(* | SNodeLit(data, rtyp)-> let d_ltyp = ltype_of_typ and
    data = L.build_call create_node_f [| data; idx |] "create_node"
builder in data *)
(* let data = L.build_malloc (ltype_of_typ styp) "data" builder in *)
(* let data = L.build_bitcast data void_ptr_t "data" builder in
    let node = L.build_call node_init_f [| idx; data|] "node_init"
builder in
    node *)

(* L.build_malloc (ltype_of_typ rtyp) "data" builder in
    let d = expr builder

    let rtyp' = L.build_malloc (ltype_of_typ rtyp) "data" builder in
    let data' = L.build_bitcast data void_ptr_t "data" builder in
    let node = L.build_call node_init_f [| data'; rtyp'|] "create_node"
builder in
    node *)
(* | SNodeData n ->
    let node = expr builder m n in
    let data_ptr = L.build_malloc (ltype_of_typ (fst e1)) "data"
builder in
    ignore (L.build_store new_value data_ptr builder);
    let data_ptr = L.build_bitcast data_ptr void_ptr_t "data"
builder in
    ignore (L.build_call node_set_data_f [| node; data_ptr |]
"node_set_data" builder);
    new_value *)
| SBinop ((A.Float,_) as e1, op, e2) ->
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
        A.Add      -> L.build_fadd
    | A.Sub       -> L.build_fsub
    | A.Mult      -> L.build_fmud
    | A.Mod       -> L.build_frem
    | A.Div       -> L.build_fdiv
    | A.Equal     -> L.build_fcmp L.Fcmp.Oeq
    | A.Neq      -> L.build_fcmp L.Fcmp.One
    | A.Less     -> L.build_fcmp L.Fcmp.Olt
    | A.Leq      -> L.build_fcmp L.Fcmp.Ole
    | A.Greater  -> L.build_fcmp L.Fcmp.Ogt

```

```

| A.Geq      -> L.build_fcmp L.Fcmp.Oge
| A.And | A.Or ->
    raise (Failure "internal error: semant should have rejected
and/or on float")
) e1' e2' "tmp" builder
| SBinop (e1, op, e2) ->
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
      A.Add      -> L.build_add
    | A.Sub      -> L.build_sub
    | A.Mult     -> L.build_mul
| A.Mod        -> L.build_srem
| A.Div        -> L.build_sdiv
    | A.And      -> L.build_and
    | A.Or       -> L.build_or
    | A.Equal    -> L.build_icmp L.Icmp.Eq
    | A.Neq     -> L.build_icmp L.Icmp.Ne
    | A.Less    -> L.build_icmp L.Icmp.Slt
    | A.Leq     -> L.build_icmp L.Icmp.Sle
    | A.Greater -> L.build_icmp L.Icmp.Sgt
    | A.Geq     -> L.build_icmp L.Icmp.Sge
    ) e1' e2' "tmp" builder
| SUnop(op, ((t, _) as e)) ->
    let e' = expr builder e in
    (match op with
      A.Neg when t = A.Float -> L.build_fneg
    | A.Neg                  -> L.build_neg
    | A.Not                  -> L.build_not) e' "tmp" builder
| SCall ("print", [e]) | SCall ("printb", [e]) ->
    L.build_call printf_func [| int_format_str ; (expr builder e) |]
    "printf" builder
| SCall ("prints", [e]) ->
    L.build_call printf_func [| string_format_str ; (expr builder e) |]
    "prints" builder
| SCall ("printbig", [e]) ->
    L.build_call printbig_func [| (expr builder e) |] "printbig"
builder
| SCall ("amalg", [e]) ->
    L.build_call amalg_func [| (expr builder e) |]
"amalg" builder
| SCall ("printf", [e]) ->

```

```

    L.build_call printf_func [| float_format_str ; (expr builder e) |]
      "printf" builder
  | SCall ("create_int_node", [e]) ->
      L.build_call create_int_node_func [| (expr builder e)
|] "create_int_node" builder
  | SCall (f, args) ->
      let (fdef, fdecl) = StringMap.find f function_decls in
      let llargs = List.rev (List.map (expr builder) (List.rev args)) in
      let result = (match fdecl.styp with
                    A.Void -> ""
                    | _ -> f ^ "_result") in
      L.build_call fdef (Array.of_list llargs) result builder
in

(* LLVM insists each basic block end with exactly one "terminator"
instruction that transfers control. This function runs "instr
builder"
if the current block does not already have a terminator. Used,
e.g., to handle the "fall off the end of the function" case. *)
let add_terminal builder instr =
  match L.block_terminator (L.insertion_block builder) with
  Some _ -> ()
  | None -> ignore (instr builder) in

(* Build the code for the given statement; return the builder for
the statement's successor (i.e., the next instruction will be built
after the one generated by this call) *)

let rec stmt builder = function
  SBlock s1 -> List.fold_left stmt builder s1
  | SExpr e -> ignore(expr builder e); builder
  | SReturn e -> ignore(match fdecl.styp with
                        (* Special "return nothing" instr *)
                        A.Void -> L.build_ret_void builder
                        (* Build return statement *)
                        | _ -> L.build_ret (expr builder e) builder );
      builder
  | SIf (predicate, then_stmt, else_stmt) ->
      let bool_val = expr builder predicate in
      let merge_bb = L.append_block context "merge" the_function in
      let build_br_merge = L.build_br merge_bb in (* partial function *)

```

```

let then_bb = L.append_block context "then" the_function in
add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
  build_br_merge;

let else_bb = L.append_block context "else" the_function in
add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
  build_br_merge;

ignore(L.build_cond_br bool_val then_bb else_bb builder);
L.builder_at_end context merge_bb

| SWhile (predicate, body) ->
let pred_bb = L.append_block context "while" the_function in
ignore(L.build_br pred_bb builder);

let body_bb = L.append_block context "while_body" the_function in
add_terminal (stmt (L.builder_at_end context body_bb) body)
  (L.build_br pred_bb);

let pred_builder = L.builder_at_end context pred_bb in
let bool_val = expr pred_builder predicate in

let merge_bb = L.append_block context "merge" the_function in
ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
L.builder_at_end context merge_bb

(* Implement for loops as while loops *)
| SFor (e1, e2, e3, body) -> stmt builder
  ( SBlock [SExpr e1 ; SWhile (e2, SBlock [body ; SExpr e3]) ] )
in

(* Build the code for each statement in the function *)
let builder = stmt builder (SBlock fdecl.sbody) in

(* Add a return if the last block falls off the end *)
add_terminal builder (match fdecl.styp with
  A.Void -> L.build_ret_void
  | A.Float -> L.build_ret (L.const_float float_t 0.0)
  | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
in

List.iter build_function_body functions;

```

```
the_module
```

```
*** Semantic ***
```

```
(* Semantic checking for the MicroC compiler *)

open Ast
open Sast

module StringMap = Map.Make(String)

(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)

let check (globals, functions) =

  (* Verify a list of bindings has no void types or duplicate names *)
  let check_binds (kind : string) (binds : bind list) =
    List.iter (function
      (Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
      | _ -> ()) binds;
    let rec dups = function
      [] -> ()
      | ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
        raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
      | _ :: t -> dups t
    in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
  in

  (**** Check global variables ****)

  check_binds "global" globals;

  (**** Check functions ****)

  (* Collect function declarations for built-in functions: no bodies *)
  let built_in_decls =
    let add_bind map (name, ty) = StringMap.add name {
      typ = Void;

```



```

    fname = name;
    formals = [(ty, "x")];
    locals = []; body = [] } map
  in List.fold_left add_bind StringMap.empty [ ("print", Int); ("prints",
String);
  ("create_int_node", Int);
                                     (* ("putc", Char); *)
  ("printb", Bool);
                                     ("printf", Float);
                                     ("printbig", Int);
("amalg",Int) ]
  in

  (* Add function name to symbol table *)
  let add_func map fd =
    let built_in_err = "function " ^ fd.fname ^ " may not be defined"
    and dup_err = "duplicate function " ^ fd.fname
    and make_err er = raise (Failure er)
    and n = fd.fname (* Name of the function *)
    in match fd with (* No duplicate functions or redefinitions of
built-ins *)
      | _ when StringMap.mem n built_in_decls -> make_err built_in_err
      | _ when StringMap.mem n map -> make_err dup_err
      | _ -> StringMap.add n fd map
  in

  (* Collect all function names into one symbol table *)
  let function_decls = List.fold_left add_func built_in_decls functions
  in

  (* Return a function from our symbol table *)
  let find_func s =
    try StringMap.find s function_decls
    with Not_found -> raise (Failure ("unrecognized function " ^ s))
  in

  let _ = find_func "main" in (* Ensure "main" is defined *)

  let check_function func =
    (* Make sure no formals or locals are void or duplicates *)
    check_binds "formal" func.formals;
    check_binds "local" func.locals;

```

```

(* Raise an exception if the given rvalue type cannot be assigned to
the given lvalue type *)
let check_assign lvaluet rvaluet err =
  if lvaluet = rvaluet then lvaluet else raise (Failure err)
in

(* Build local symbol table of variables for this function *)
let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name ty
m)
  StringMap.empty (globals @ func.formals @ func.locals
)
in

(* Return a variable from our local symbol table *)
let type_of_identifer s =
  try StringMap.find s symbols
  with Not_found -> raise (Failure ("undeclared identifier " ^ s))
in

(* Return a semantically-checked expression, i.e., with a type *)
let rec expr = function
  Literal l -> (Int, SLiteral l)
  | SLiteral l -> (String, SSLiteral l)
  (* | Cliteral l -> (Char, SCliteral l) *)
  | Fliteral l -> (Float, SFliteral l)
  | BoolLit l -> (Bool, SBoolLit l)
  (* | NodeLit (e1,e2) -> let (e,t) = expr symbols e1 in
  (SNode t, SNodeLit (check_node_index (expr symbols e1), (t,e))) *)
  | Noexpr -> (Void, SNoexpr)
  | Id s -> (type_of_identifer s, SId s)
  | Assign(var, e) as ex ->
    let lt = type_of_identifer var
    and (rt, e') = expr e in
    let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
      string_of_typ rt ^ " in " ^ string_of_expr ex
    in (check_assign lt rt err, SAssign(var, (rt, e')))
  | Unop(op, e) as ex ->
    let (t, e') = expr e in
    let ty = match op with
      Neg when t = Int || t = Float -> t
    | Not when t = Bool -> Bool

```

```

    | _ -> raise (Failure ("illegal unary operator " ^
                          string_of_uop op ^ string_of_typ t ^
                          " in " ^ string_of_expr ex))
    in (ty, SUnop(op, (t, e'))))
| Binop(e1, op, e2) as e ->
  let (t1, e1') = expr e1
  and (t2, e2') = expr e2 in
  (* All binary operators require operands of the same type *)
  let same = t1 = t2 in
  (* Determine expression type based on operator and operand types
*)

  let ty = match op with
    Add | Sub | Mult | Mod | Div when same && t1 = Int -> Int
  | Add | Sub | Mult | Mod | Div when same && t1 = Float -> Float
  | Equal | Neq          when same -> Bool
  | Less | Leq | Greater | Geq
    when same && (t1 = Int || t1 = Float) -> Bool
  | And | Or when same && t1 = Bool -> Bool
  | _ -> raise (
    Failure ("illegal binary operator " ^
            string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
            string_of_typ t2 ^ " in " ^ string_of_expr e))
  in (ty, SBinop((t1, e1'), op, (t2, e2'))))
| Call(fname, args) as call ->
  let fd = find_func fname in
  let param_length = List.length fd.formals in
  if List.length args != param_length then
    raise (Failure ("expecting " ^ string_of_int param_length ^
                    " arguments in " ^ string_of_expr call))
  else let check_call (ft, _) e =
    let (et, e') = expr e in
    let err = "illegal argument found " ^ string_of_typ et ^
              " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e
    in (check_assign ft et err, e')
  in
  let args' = List.map2 check_call fd.formals args
  in (fd.typ, SCall(fname, args'))
in

let check_bool_expr e =
  let (t', e') = expr e
  and err = "expected Boolean expression in " ^ string_of_expr e

```

```

    in if t' != Bool then raise (Failure err) else (t', e')
  in

  (* Return a semantically-checked statement i.e. containing sexprs *)
  let rec check_stmt = function
    Expr e -> SExpr (expr e)
  | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt
b2)
  | For(e1, e2, e3, st) ->
    SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
  | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
  | Return e -> let (t, e') = expr e in
    if t = func.typ then SReturn (t, e')
    else raise (
      Failure ("return gives " ^ string_of_typ t ^ " expected " ^
        string_of_typ func.typ ^ " in " ^ string_of_expr e))

    (* A block is correct if each statement is correct and nothing
    follows any Return statement. Nested blocks are flattened. *)
  | Block s1 ->
    let rec check_stmt_list = function
      [Return _ as s] -> [check_stmt s]
    | Return _ :: _ -> raise (Failure "nothing may follow a
return")
  | Block s1 :: ss -> check_stmt_list (s1 @ ss) (* Flatten
blocks *)
  | s :: ss -> check_stmt s :: check_stmt_list ss
  | [] -> []
    in SBlock(check_stmt_list s1)

  in (* body of check_function *)
  { styp = func.typ;
    sfname = func.fname;
    sformals = func.formals;
    slocals = func.locals;
    sbody = match check_stmt (Block func.body) with
      SBlock(s1) -> s1
    | _ -> raise (Failure ("internal error: block didn't become a
block?"))
  }
  in (globals, List.map check_function functions)

```

Top-Level

```
(* Top-level of the MicroC compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR,
   and dump the module *)
```

```
type action = Ast | Sast | LLVM_IR | Compile
```

```
let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
     "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./microc.native [-a|-s|-l|-c] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename)
  usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = Microcparse.program Scanner.token lexbuf in
  match !action with
  | Ast -> print_string (Ast.string_of_program ast)
  | _ -> let sast = Semant.check ast in
  match !action with
  | Ast -> ()
  | Sast -> print_string (Sast.string_of_sprogram sast)
  | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate
sast))
  | Compile -> let m = Codegen.translate sast in
  Llvm_analysis.assert_valid_module m;
  print_string (Llvm.string_of_llmodule m)
```

*** Makefile***

```
(* Code generation: translate takes a semantically checked AST and
   produces LLVM IR
```

LLVM tutorial: Make sure to read the OCaml version of the tutorial

<http://llvm.org/docs/tutorial/index.html>

Detailed documentation on the OCaml LLVM library:

<http://llvm.moe/>

<http://llvm.moe/ocaml/>

newversion

*)

```
module L = Llvm
```

```
module A = Ast
```

```
open Sast
```

```
module StringMap = Map.Make(String)
```

```
(* translate : Sast.program -> Llvm.module *)
```

```
let translate (globals, functions) =
```

```
  let context    = L.global_context () in
```

```
  (* Create the LLVM compilation module into which  
  we will generate code *)
```

```
(* let llmem = L.MemoryBuffer.of_file "tree.bc" in
```

```
let llm = Llvm_bitreader.parse_bitcode context llmem in *)
```

```
let the_module = L.create_module context "MicroC" in
```

```
(* Get types from the context *)
```

```
let i32_t      = L.i32_type    context
```

```
and i8_t      = L.i8_type     context
```

```
and i1_t      = L.i1_type     context
```

```
and str_t     = L.pointer_type (L.i8_type context)
```

```
and float_t   = L.double_type context
```

```
and void_ptr_t = L.pointer_type (L.i8_type context)
```

```
and void_t    = L.void_type   context
```

```
(* and node_t    = L.pointer_type (match L.type_by_name llm "struct.Node"  
with
```

```
  None -> raise (Failure "Missing implementation for struct Node")
```

```
  | Some t -> t) in *)
```

```
in
```

```
(* Return the LLVM type for a MicroC type *)
```

```
let ltype_of_typ = function
```

```

    A.Int    -> i32_t
  (*      | A.Char  -> i8_t *)
  | A.String -> str_t
  | A.Bool   -> i1_t
  | A.Float  -> float_t
  | A.Void   -> void_t
  (* | A.Node(_) -> node_t *)
  in
  (*
  (* node functions *)
  let create_node_t = L.function_type node_t [| void_ptr_t ; i32_t |] in
  let create_node_f = L.declare_function "create_node" create_node_t
the_module in
  let root_init_t = L.function_type node_t [| node_t |] in
  let root_init_f = L.declare_function "root_init" root_init_t the_module
  in
  let delete_node_t = L.function_type void_ptr_t [| node_t |] in
  let delete_node_f = L.declare_function "delete_node" delete_node_t
the_module in
  let add_child_t = L.function_type node_t [| node_t; node_t|] in
  let add_child_f = L.declare_function "add_child" add_child_t the_module
  in
  let get_root_t = L.function_type node_t [| node_t |] in
  let get_root_f = L.declare_function "get_root" get_root_t the_module in
  let is_ancestor_t = L.function_type i32_t [| node_t ; node_t |] in
  let is_ancestor_f = L.declare_function "is_ancestor" is_ancestor_t
the_module in
  let is_root_t = L.function_type i32_t [| node_t |] in
  let is_root_f = L.declare_function "is_root" is_root_t the_module in
  let get_depth_t = L.function_type i32_t [| node_t |] in
  let get_depth_f = L.declare_function "get_depth" get_depth_t the_module
  in
  let shift_right_t = L.function_type void_t [| i32_t; node_t |] in
  let shift_right_f = L.declare_function "shift_right" shift_right_t
the_module in
  let shift_left_t = L.function_type void_t [| i32_t; node_t |] in
  let shift_left_f = L.declare_function "shift_left" shift_left_t
the_module in
  *)
  (* Create a map of global variables after creating each *)
  let global_vars : L.lvalue StringMap.t =

```

```

let global_var m (t, n) =
  let init = match t with
    | A.Float -> L.const_float (ltype_of_typ t) 0.0
    | _ -> L.const_int (ltype_of_typ t) 0
  in StringMap.add n (L.define_global n init the_module) m in
List.fold_left global_var StringMap.empty globals in

let printf_t : L.lltype =
  L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let printf_func : L.llvalue =
  L.declare_function "printf" printf_t the_module in

let create_int_node : L.lltype =
  L.var_arg_function_type i32_t [| i32_t |] in
let create_int_node_func : L.llvalue =
  L.declare_function "create_int_node" create_int_node the_module
in
let amalg : L.lltype =
  L.function_type i32_t [| i32_t |] in
let amalg_func : L.llvalue =
  L.declare_function "amalg" amalg the_module in

let printbig_t : L.lltype =
  L.function_type i32_t [| i32_t |] in
let printbig_func : L.llvalue =
  L.declare_function "printbig" printbig_t the_module in

(* Define each function (arguments and return type) so we can
   call it even before we've created its body *)
let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
  let function_decl m fdecl =
    let name = fdecl.sfname
      and formal_types =
        Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.sformals)
    in let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types
  in
  StringMap.add name (L.define_function name ftype the_module, fdecl) m
in
List.fold_left function_decl StringMap.empty functions in

(* Fill in the body of the given function *)
let build_function_body fdecl =

```



```

let (the_function, _) = StringMap.find fdecl.sfname function_decls in
let builder = L.builder_at_end context (L.entry_block the_function) in

let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder
and string_format_str = L.build_global_stringptr "%s\n" "fmt" builder
in
  (* and char_format_str = L.build_global_stringptr "%c\n" "fmt" builder
in *)

  (* Construct the function's "locals": formal arguments and locally
  declared variables. Allocate each on the stack, initialize their
  value, if appropriate, and remember their values in the "locals" map
  *)
  let local_vars =
    let add_formal m (t, n) p =
      L.set_value_name n p;
      let local = L.build_alloca (ltype_of_typ t) n builder in
      ignore (L.build_store p local builder);
      StringMap.add n local m
    and add_local m (t, n) =
      let local_var = L.build_alloca (ltype_of_typ t) n builder
      in StringMap.add n local_var m
    in

    let formals = List.fold_left2 add_formal StringMap.empty
fdecl.sformals
      (Array.to_list (L.params the_function)) in
      List.fold_left add_local formals fdecl.slocals
    in

  (* Return the value for a variable or formal argument.
  Check local names first, then global names *)
  let lookup n = try StringMap.find n local_vars
    with Not_found -> StringMap.find n global_vars
  in

  (* Construct code for an expression; return its value *)
  let rec expr builder ((_, e) : sexpr) = match e with

```

```

SLiteral i  -> L.const_int i32_t i
| SSLiteral s -> L.build_global_stringptr s "str" builder
(*
  | SCliteral c -> L.build_global_stringptr c "str" builder *)
| SBoolLit b  -> L.const_int i1_t (if b then 1 else 0)
| SFliteral l -> L.const_float_of_string float_t l
| SNoexpr     -> L.const_int i32_t 0
| SId s       -> L.build_load (lookup s) s builder
| SAssign (s, e) -> let e' = expr builder e in
                    ignore(L.build_store e' (lookup s) builder); e'
(* | SNodeLit(data, rtyp)-> let d_ltyp = ltype_of_typ and
  data = L.build_call create_node_f [| data; idx |] "create_node"
builder in data *)
(* let data = L.build_malloc (ltype_of_typ styp) "data" builder in *)
(* let data = L.build_bitcast data void_ptr_t "data" builder in
  let node = L.build_call node_init_f [| idx; data|] "node_init"
builder in
  node *)

(* L.build_malloc (ltype_of_typ rtyp) "data" builder in
  let d = expr builder

  let rtyp' = L.build_malloc (ltype_of_typ rtyp) "data" builder in
  let data' = L.build_bitcast data void_ptr_t "data" builder in
  let node = L.build_call node_init_f [| data'; rtyp'|] "create_node"
builder in
  node *)
(* | SNodeData n ->
  let node = expr builder m n in
  let data_ptr = L.build_malloc (ltype_of_typ (fst e1)) "data"
builder in
  ignore (L.build_store new_value data_ptr builder);
  let data_ptr = L.build_bitcast data_ptr void_ptr_t "data"
builder in
  ignore (L.build_call node_set_data_f [| node; data_ptr |]
"node_set_data" builder);
  new_value *)
| SBinop ((A.Float,_) as e1, op, e2) ->
  let e1' = expr builder e1
  and e2' = expr builder e2 in
  (match op with
    A.Add      -> L.build_fadd
  | A.Sub      -> L.build_fsub

```

```

    | A.Mult    -> L.build_fmuls
  | A.Mod      -> L.build_frem
    | A.Div     -> L.build_fdiv
    | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
    | A.Neq     -> L.build_fcmp L.Fcmp.One
    | A.Less    -> L.build_fcmp L.Fcmp.Olt
    | A.Leq     -> L.build_fcmp L.Fcmp.Ole
    | A.Greater -> L.build_fcmp L.Fcmp.Ogt
    | A.Geq     -> L.build_fcmp L.Fcmp.Oge
    | A.And | A.Or ->
      raise (Failure "internal error: semant should have rejected
and/or on float")
    ) e1' e2' "tmp" builder
  | SBinop (e1, op, e2) ->
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
      A.Add      -> L.build_add
    | A.Sub      -> L.build_sub
    | A.Mult     -> L.build_mul
  | A.Mod       -> L.build_srem
  | A.Div       -> L.build_sdiv
    | A.And      -> L.build_and
    | A.Or       -> L.build_or
    | A.Equal    -> L.build_icmp L.Icmp.Eq
    | A.Neq     -> L.build_icmp L.Icmp.Ne
    | A.Less    -> L.build_icmp L.Icmp.Slt
    | A.Leq     -> L.build_icmp L.Icmp.Sle
    | A.Greater -> L.build_icmp L.Icmp.Sgt
    | A.Geq     -> L.build_icmp L.Icmp.Sge
    ) e1' e2' "tmp" builder
  | SUnop(op, ((t, _) as e)) ->
    let e' = expr builder e in
    (match op with
      A.Neg when t = A.Float -> L.build_fneg
    | A.Neg                  -> L.build_neg
    | A.Not                  -> L.build_not) e' "tmp" builder
  | SCall ("print", [e]) | SCall ("printb", [e]) ->
    L.build_call printf_func [| int_format_str ; (expr builder e) |]
    "printf" builder
  | SCall ("prints", [e]) ->
    L.build_call printf_func [| string_format_str ; (expr builder e) |]

```

```

    "prints" builder
    | SCall ("printbig", [e]) ->
        L.build_call printbig_func [| (expr builder e) |] "printbig"
builder
    | SCall ("amalg", [e]) ->
        L.build_call amalg_func [| (expr builder e) |]
"amalg" builder
    | SCall ("printf", [e]) ->
        L.build_call printf_func [| float_format_str ; (expr builder e) |]
        "printf" builder
    | SCall ("create_int_node", [e]) ->
        L.build_call create_int_node_func [| (expr builder e)
]|] "create_int_node" builder
    | SCall (f, args) ->
        let (fdef, fdecl) = StringMap.find f function_decls in
        let llargs = List.rev (List.map (expr builder) (List.rev args)) in
        let result = (match fdecl.styp with
            A.Void -> ""
            | _ -> f ^ "_result") in
        L.build_call fdef (Array.of_list llargs) result builder
in

(* LLVM insists each basic block end with exactly one "terminator"
instruction that transfers control. This function runs "instr
builder"
if the current block does not already have a terminator. Used,
e.g., to handle the "fall off the end of the function" case. *)
let add_terminal builder instr =
    match L.block_terminator (L.insertion_block builder) with
    Some _ -> ()
    | None -> ignore (instr builder) in

(* Build the code for the given statement; return the builder for
the statement's successor (i.e., the next instruction will be built
after the one generated by this call) *)

let rec stmt builder = function
    SBlock sl -> List.fold_left stmt builder sl
    | SExpr e -> ignore(expr builder e); builder
    | SReturn e -> ignore(match fdecl.styp with
        (* Special "return nothing" instr *)
        A.Void -> L.build_ret_void builder

```

```

                (* Build return statement *)
                | _ -> L.build_ret (expr builder e) builder );
            builder
| SIf (predicate, then_stmt, else_stmt) ->
    let bool_val = expr builder predicate in
    let merge_bb = L.append_block context "merge" the_function in
    let build_br_merge = L.build_br merge_bb in (* partial function *)

    let then_bb = L.append_block context "then" the_function in
    add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
        build_br_merge;

    let else_bb = L.append_block context "else" the_function in
    add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
        build_br_merge;

    ignore(L.build_cond_br bool_val then_bb else_bb builder);
    L.builder_at_end context merge_bb

| SWhile (predicate, body) ->
    let pred_bb = L.append_block context "while" the_function in
    ignore(L.build_br pred_bb builder);

    let body_bb = L.append_block context "while_body" the_function in
    add_terminal (stmt (L.builder_at_end context body_bb) body)
        (L.build_br pred_bb);

    let pred_builder = L.builder_at_end context pred_bb in
    let bool_val = expr pred_builder predicate in

    let merge_bb = L.append_block context "merge" the_function in
    ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
    L.builder_at_end context merge_bb

(* Implement for loops as while loops *)
| SFor (e1, e2, e3, body) -> stmt builder
    ( SBlock [SEExpr e1 ; SWhile (e2, SBlock [body ; SEExpr e3]) ] )
in

(* Build the code for each statement in the function *)
let builder = stmt builder (SBlock fdecl.sbody) in

```

```

(* Add a return if the last block falls off the end *)
add_terminal builder (match fdecl.styp with
  A.Void -> L.build_ret_void
  | A.Float -> L.build_ret (L.const_float float_t 0.0)
  | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
in

List.iter build_function_body functions;
the_module

```

Appendix: Original Tree++ Code

Scanner

```

(* Authors:
Allison Costa arc2211
Laura Matos lm3081
Jacob Penn jp3666
Laura Smerling les2206
*)
(* Ocamllex scanner for Tree++ *)
{ open Parser }

(* Definitions *)
let digit = ['0'-'9']
let decimal = ((digit+'.'digit*)|('.'digit+))
let letter = ['a'-'z' 'A'-'Z']

(* Rules *)
rule token = parse
  [ ' ' '\t' '\r' '\n' ] { token lexbuf } (* to ignore whitespace *)
  | "(" { comment lexbuf }
  | '.' { DOT }
  | ',' { COMMA }

  (* scoping *)
  | '(' { LPAREN }
  | ')' { RPAREN }
  | '{' { LBRACE }

```

```
| '}'      { RBRACE }
| '['      { LBRACK }
| ']'      { RBRACK }
```

```
(* keywords *)
```

```
(* | "bfs" {BFS}
| "dfs" {DFS} these have been classified as library functions*)
| "if" { IF }
| "then" { THEN }
| "else" { ELSE }
| "for" { FOR }
| "float" { FLOAT }
| "double" { DOUBLE }
| "int" { INT }
| "string" { STRING }
| "tuple" { TUPLE }
| "node" { NODE }
| "parent" { PARENT }
| "data" { DATA }
| "null" { NULL }
| "return" { RETURN }
| "break" { BREAK }
| "range" { RANGE }
| "func" { FUNCTION }
```

```
(* operators *)
```

```
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
(* | "+=" { PLUSEQ }
| "-=" { MINUSEQ }
| "*=" { TIMESEQ }
| "/=" { DIVEQ } to be included in next vertical slice? *)
| '%'      { MOD }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "=="     { EQ }
| ">"      { GT }
| "<"      { LT }
| "!="     { NEQ }
```

```

| ">=" { GEQ }
| "<=" { LEQ }
| ">>" { RSHIFT }
| "<<" { LSHIFT }
| "<<+" { LSHIFTPLUS }
| ">>+" { RSHIFTPLUS }
| "<<- " { LSHIFTMINUS }
| ">>- " { RSHIFTMINUS }
| "^^" { DSWAP }
| ':' { BFSNODE }

```

```

(* literals and IDs *)
| digit+ as lxm { INT_LIT(int_of_string lxm)
}
| decimal as lxm { FLOAT_LIT(float_of_string
lxm) }
| ("true" | "false") as lxm { BOOL_LIT(bool_of_string
lxm) }
| '\"' ([^'\']* as lxm) '\"' { STRING_LIT(lxm) }
| eof { EOF }
| letter (letter | digit | '_')* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

```

```

and comment =
  parse "*" { token lexbuf } (* return to normal scanning *)
  | _ { comment lexbuf } (* Ignore other character *)

```

Parser

```

/* PARSER Authors: */
%{ open AST %}

```

```

%token PLUS MINUS TIMES DIVIDE ASSIGN
%token DOT COLON SEMI COMMA
%token LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
%token IF THEN ELSE FOR BOOL FLOAT INT VOID STRING TUPLE NODE ROOT DATA

```



```

PARENT NULL RETURN BREAK RANGE FUNCTION
%token MOD AND OR NOT LT GT EQ NEQ GEQ LEQ RSHIFT LSHIFT LSHIFTPLUS
RSHIFTPLUS LSHIFTMINUS RSHIFTMINUS DSWAP BFSNODE
%token <int> INT_LIT
%token <bool> BOOL_LIT
%token <string> ID STRING_LIT FLOAT_LIT
%token EOF

%start program
%type <Ast.program> program

%right ASSIGN
%right NOT NEG
%left DOT
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left OR AND
%left SWITCH
%left RSHIFT LSHIFT LSHIFTPLUS RSHIFTPLUS LSHIFTMINUS RSHIFTMINUS DSWAP
BFSNODE

%right EQ NEQ
%nonassoc LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
%nonassoc GEQ LEQ LT GT

%%
program:
    stmt_list EOF {List.rev $1}

stmt_list:
    /* empty */ { [] }
| stmt_list stmt { $2 :: $1 }

typ:
    INT { Int }
| BOOL { Bool }
| FLOAT { Float }
| VOID { Void }
| STRING { String }
| TUPLE LT typ GT { Tuple($3) }
| NODE LT typ GT { Node($3) }
| FUNCTION LT typ GT { Function($3) }

```

```

/* return must return something */

stmt:
    expr SEMI                {Expr $1}
  | RETURN expr SEMI        {Return $2}
  | LBRACE stmt_list RBRACE  {Block(list.rev $2) }
  | IF LPAREN expr RPAREN THEN stmt ELSE stmt { If($3, $6, $8)      }
  | FOR LPAREN expr SEMI expr SEMI expr RPAREN LBRACE stmt RBRACE { For($3,
$5, $7, $10)  }
  | typ ID ASSIGN expr SEMI { Dcl ($1,$2, $4) }

/* Currently does not allow declarations without assignments */

expr:
    INT_LIT      {Ilit($1) }
  | FLOAT_LIT    {Flit($1) }
  | BOOL_LIT     {Blit($1) }
  | STRING_LIT   {Slit($1) }
  | NULL         {Null     }
/* Arithmetics */
  | NOT expr     { Unop(Not, $2)      }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | expr PLUS expr { Binop($1, Add, $3) }
  | expr MINUS expr { Binop($1, Sub, $3) }
  | expr TIMES expr { Binop($1, Mult, $3) }
  | expr DIVIDE expr { Binop($1, Div, $3) }
  | expr MOD expr { Binop($1, Mod, $3) }
  | expr EQ expr { Binop($1, Equal, $3) }
  | expr NEQ expr { Binop($1, Neq, $3) }
  | expr LT expr { Binop($1, Less, $3) }
  | expr LEQ expr { Binop($1, Leq, $3) }
  | expr GT expr { Binop($1, Greater, $3) }
  | expr GEQ expr { Binop($1, Geq, $3) }
  | expr AND expr { Binop($1, And, $3) }
  | expr OR expr { Binop($1, Or, $3) }
/* | LPAREN expr RPAREN { $2 } */
/*Node Operations*/
  | expr RSHIFT { Unop(Rshiftu, $1)      }
  | expr LSHIFT { Unop(Lshiftu, $1)      }
  | expr RSHIFT LBRACK expr RBRACK { Binop($1, Rshiftb, $4)      }

```

```

| expr LSHIFT LBRACK expr RBRACK          { Binop($1, Lshiftb, $4)      }
| expr LSHIFTPLUS  { Unop(Lshiftplus, $1)  }
| expr RSHIFTPLUS  { Unop(Rshiftplus, $1)  }
| expr LSHIFTMINUS { Unop(Lshiftminus, $1) }
| expr RSHIFTMINUS { Unop(Rshiftminus, $1) }
| expr DSWAP expr  { Binop($1, Dswap, $3)  }
| expr BFSNODE expr { Binop($1, Bfsnode, $3) }
/*Assignment*/
| ID          {          Id($1)          }
| ID ASSIGN expr { Assign(Id($1), $3)    }

/* Node Auxillary */
| LBRACK expr COMMA expr RBRACK  { NodeLit($2,$4)  }
| expr DOT DATA                  { NodeData($1)    }
| expr DOT DATA ASSIGN expr     { Assign(NodeData($1),$5) }
| expr DOT PARENT                 { Parent($1)}
| expr LBRACE expr RBRACE        { NodeChild($1,$3)}

/* Tuple */

| tuple_list          {$1          }
| access              {$1          }
| access ASSIGN expr {Assign($1, $3) }

/* Functions */
| funcexpr          { FuncExpr($1) }
| fn_call           { $1   }

/* TUPLE AUXILLARY */

/* access to tuple elements can be recursively called */
access:
    ID LBRACK expr RBRACK          {Index($1,$3)}
    | access LBRACK expr RBRACK    {Indexr($1,$3)}

fn_call:
    ID LPAREN args_opt RPAREN { Call ($1, $3)  }

/* Functions - return type cannot be omitted */

```

```

funcexpr:
    FUNCTION LT typ GT LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
    { { typ = $3;
      formals = $6;
      body = List.rev $9} }

```

```

formals_opt:
    /*nothing*/ { [] }
    | formal_list { List.rev $1 }

```

```

formal_list:
    typ ID { [($1,$2)] }
    | formal_list COMMA typ ID { ($3,$4) :: $1 }

```

```

tuple_list:
    | LPAREN elems_seq_opt RPAREN { TupleLit(List.rev $2) }

```

```

elems_seq_opt:
    /*nothing*/ { [] }
    | expr { [$1] }
    | elems_seq_opt COMMA expr { $3 :: $1 }

```

/* Function call auxiliary patterns */

```

args_opt:
    /* nothing */ { [] }
    | args_list { List.rev $1 }

```

```

args_list:
    expr { [$1] }
    | args_list COMMA expr { $3 :: $1 }

```

Semantic

(* Semantically-checked Abstract Syntax Tree **and** functions **for** printing **it** *)

```
open Ast
```

```
type sexpr = typ * sx
```

```
and sx =
```

```
  SIlit of int
  | SFlit of string
  | SBlit of bool
  | SSlit of string
  | SNodeLit of sexpr * sexpr
  | SNodeData of sexpr
  | SNodeChild of sexpr * sexpr
  | SParent of sexpr
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SAssign of sexpr * sexpr
  | SCall of string * sexpr list
  | STupleLit of sexpr list
  | SId of string
  | SIndex of string * sexpr
  | SIndextr of expr * sexpr
  | SFuncExpr of sfunc_decl
  | SNull
  | SNoexpr
```

```
type sstmt =
```

```
  SBlock of sstmt list
  | SDcl of styp * string * sexpr
  | SExpr of sexpr
  | SReturn of sexpr
  | SBfsnode of string * sexpr * sexpr * sstmt
  | SIf of sexpr * sstmt * sstmt
  | SThen of sstmt
  | SElse of sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
```

```
type sfunc_decl = {
```

```
  styp : typ;
  sfname : string;
  sformals : bind list;
  slocals : bind list;
  sbody : sstmt list;
}
```

```

type sprogram = bind list * sfunc_decl list

(* Pretty-printing functions *)

let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (match e with
    SIlit(l) -> string_of_int l
  | SBlit(true) -> "true"
  | SBlit(false) -> "false"
  | SFlit(l) -> l
  | SSlit(l) -> l ^ ")")
  | SNodeLit(l, e) -> string_of_sexpr l ^ "#" ^ string_of_sexpr e ^ ")"
  | SNodeData e -> "(" ^ string_of_sexpr e ^ ".data)"
  | SId(s) -> s
  | SBinop(e1, o, e2) ->
    string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
  | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr e
  | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
  | SCall(f, e1) ->
    f ^ "(" ^ String.concat ", " (List.map string_of_sexpr e1) ^ ")"
  | SIndex(id, i) -> id ^ "[" ^ string_of_sexpr i ^ "]" ^ ")"
  | SIndexr(i,j) -> string_of_sexpr i ^ "[" ^ string_of_sexpr j ^ "]" ^ ")"
  | SFuncExpr(fexpr) -> string_of_sfexpr fexpr ^ ")"
  | SNoexpr -> ""
    ) ^ ")"
  | SNull -> "NULL"

let rec string_of_sstmt = function
  SBlock(stmts) ->
    "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
  | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
  | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
  | SIf(e, s, SBlock([])) ->
    "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
  | SIf(e, s1, s2) -> "if (" ^ string_of_sexpr e ^ ")\n" ^
    string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SFor(e1, e2, e3, s) ->
    "for (" ^ string_of_sexpr e1 ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
    string_of_sexpr e3 ^ ") " ^ string_of_sstmt s
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt
s

```

```

| SDcl(t, id, (SVoid,SNoexpr)) -> string_of_styp t ^ " " ^ id ^ ";\n"
| SDcl(t, id, v) -> string_of_styp t ^ " " ^ id ^ " = " ^ string_of_sexpr
v ^ ";\n"

```

```

let string_of_sfdecl fdecl =
  string_of_typ fdecl.styp ^ " " ^
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sformals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.slocals) ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "}\n"

```

```

let string_of_sprogram (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_sfdecl funcs)

```

Semantic

(* Semantic checking for the treePlusPlus compiler *)

open Ast

module StringMap = Map.Make(String)

(* Semantic checking of a program. Returns void if successful,
throws an exception if something is wrong.
Check each global variable, then check each function *)

let check (globals, function) =

(*Raise an exception if the given list has duplicate *)

let report_duplicate exceptf list =

let rec helper = function

n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))

| _ :: t -> helper t

| [] -> ()

in helper (List.sort compare list)

in

(* Raise an exception if a given binding is to a void type*)

```

let check_not_void exceptf = function
  (Void, n) -> raise (Failure (exceptf n))
  |_-> ()
in

(* Raise an exception of the given rvalue type cannot be assigned to
the given lvalue type *)
let check_assign lvaluet rvaluet err =
  if (lvaluet) = (rvaluet) then lvaluet else raise err
in

(*can add in other versions of assigning here*)

(**** Checking Global Variables ****)

List.iter (check_not_void (fun n -> "illegal void global " ^ n)) globals;
report_duplicate (fun n -> "duplicate global " ^ n) (List.map snd
globals);

(**** Checking Functions ****)

if List.mem "print" (List.map (fun fd -> fd.fname) functions)
then raise (Failure ("function print may not be defined")) else ();

report_duplicate (fun n -> "duplicate function " ^ n)
(List.map ( fun fd -> fd.fname) functions);

(*Function declaration for a named function*)
let built_in_decls = StringMap.add "print"
  { typ = Void; fname = "print"; formals = [(Int, "x")];
    locals = []; body = [] }
in

let function_decls = List.fold_left (fun m fd -> Stringmap.add fd.fname
fd m)
  built_in_decls functions
in

let function_decl s = try StringMap.find s function_decls
  with Not_found -> raise (Failure ("unrecognized function " ^ s))

```



```

in

(*main check*)
let _ = function_decl "main" in (*Ensure "main" is defined *)

(*add in all the function declirations, linear*)

let check_function func =
  List.iter (check_not_void (fun n -> "illegal void formal " ^ n ^
    " in " func.fname)) func.formals;

  report_duplicate (fun n -> "duplicate formal " ^ n ^ " in " ^
func.fname)
  (List.map snd func.locals);

  (*Type of each variable (global, formal, or local) *)
  let symbol = List.fold_left (fun m (t, n) -> StringMap.add n t m)
    StringMap.empty
    (globals @ func.formals @ func.locals )
in

let type_of_identifier s =
  try StringMap.find s symbols
  with Not_found ->raise (Failure ("undeclared identifier " ^ s))
in

(* get types for node *)
let getNodeType = function
  NodeType(typ) -> typ
  | _ -> Void
in

let getTupleType = function
  TupleType(typ) -> typ
  | _ -> String
in

(* Return the type of an expression or throw an exception *)
let rec expr = function
  IntLit _ -> Int
  | BoolLit _ -> Bool
  | FloatLit _ -> Float

```

```

| StringLit _ -> String
| Tuple (t,_) -> TupleType(t)
| Node(_, t) -> NodeType(t)
| Id s -> type_of_identifier s
| Binop(e1, op, e2) as e -> let t1 = expr e1
                           and t2 = expr e2 in
  (match op with
    Add      when t1 = Float && t2 = Float -> Float
  | Add      when t1 = Int && t2 = Int -> Int
  | Add      when t1 = Bool && t2 = Bool -> Bool
  | Div      when t1 = Float && t2 = Float -> Float
  | Div      when t1 = Int && t2 = Int -> Int
  | Equal    when t1 = t2 -> Bool
  | Geq     when t1 = Float && t2 = Float -> Bool
  | Geq     when t1 = Int && t2 = Int -> Bool
  | Greater  when t1 = Float && t2 = Float -> Bool
  | Greater  when t1 = Int && t2 = Int -> Bool
  | Leq     when t1 = Float && t2 = Float -> Bool
  | Leq     when t1 = Int && t2 = Int -> Bool
  | Less    when t1 = Float && t2 = Float -> Bool
  | Less    when t1 = Int && t2 = Float -> Bool
  | Mult    when t1 = Float && t2 = Float -> Float
  | Mult    when t1 = Int && t2 = Int -> Int
  | Neq     when t1 = t2 -> Bool
  | Or      when t1 = Bool && t2 = Bool -> Bool
  | Sub     when t1 = Float && t2 = Float -> Float
  | Sub     when t1 = Int && t2 = Int -> Int
  | _ -> raise (Failure ("illegal binary operator " ^
                        string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                        string_of_typ t2 ^ " in " ^ string_of_expr e)))
| Unop(op, e) as ex -> let t = expr e in
  (match op with
    Neg when t = Int -> Int
  | Not when t = Bool -> Bool
  | _ -> raise (Failure ("illegal unary operator " ^ string_of_uop op
                        string_of_typ t ^ " in " ^ string_of_expr ex)))
| Noexpr -> Void

```

^

Appendix: Other Info

Project Log Based On GitHub:

To illustrate our contributions to the project:

```
git log --all --decorate --oneline --graph
```

```
graph TD
  a83aa3e[a83aa3e with funcall and funcdec] --> 784e04d[784e04d working parser, no errors]
  784e04d --> b63c36c[b63c36c parse with func decl]
  b63c36c --> 8144763[8144763 improvements, parser not ready]
  8144763 --> 9f5eb07[9f5eb07 changed semant so that our program is defined as a list of statements]
  9f5eb07 --> 3fcdece[3fcdece added the modified ast]
  3fcdece --> 45beaca[45beaca (node) some basic semantic stuff but not finished]
  45beaca --> 970bc9f[970bc9f (origin/node) Merge branch 'node' of https://github.com/jacobmpenn/Treep into node]
  970bc9f --> b254cec[b254cec fixed syntax errors in ast and sast]
  b254cec --> 66aa403[66aa403 shanges node to expr]
  66aa403 --> 9f8c9c7[9f8c9c7 fixed scanner and microparse no shift reduce errors]
  9f8c9c7 --> 3dca292[3dca292 Merge branch 'node' of https://github.com/jacobmpenn/Treep into node]
  3dca292 --> c630f7c[c630f7c added node to scanner and parser, no shift reduce errors]
  c630f7c --> 05a476a[05a476a scanner node]
  05a476a --> 27d6a90[27d6a90 added node - may have errors ast.ml sast.ml]
  27d6a90 --> b962552[b962552 added node to ast.ml, may have bugs]
  b962552 --> d42ffe8[d42ffe8 (stringPlus) no shift reduce]
  d42ffe8 --> 9688cbf[9688cbf hello world]
  9688cbf --> 860a173[860a173 errors]
  860a173 --> b18d064[b18d064 still not working for string]
  b18d064 --> 1439ec3[1439ec3 still not working string]
  1439ec3 --> e42b742[e42b742 still not working for string]
  e42b742 --> b9ab300[b9ab300 not working string]
  b9ab300 --> fe59858[fe59858 mod working]
  fe59858 --> ff552a1[ff552a1 (origin/dev) test]
  ff552a1 --> ac34f89[ac34f89 (dev) without nodes]
  ac34f89 --> 8037bde[8037bde (origin/asn, asn) working on assignment]
  8037bde --> 66dc0df[66dc0df (origin/ops) added node ops to all files except codegen, untested pending node completion]
  66dc0df --> 03fcc00[03fcc00 test for nodes]
  03fcc00 --> 6d475af[6d475af added nodes but still with error in codegen.ml]
  6d475af --> b3682fe[b3682fe mod, string, correct comments, no char]
  b3682fe --> 15bad77[15bad77 mod, char, string, lbrack, rbrack]
  15bad77 --> 7218329[7218329 added tests, now works for mod and print hello]
  7218329 --> 88e927e[88e927e submit hello world works for mod]
  88e927e --> 251bb9[251bb9 added the printc and prints]
  251bb9 --> 5074b14[5074b14 added mod, string, char (no errors but not tested)]
  5074b14 --> fb14440[fb14440 bug]
```

```

* | d429a16 Merge branch 'decl' of https://github.com/jacobmpenn/Trepp into decl
* | f35567b bunch of edits msotly to epress sast form
* | 222510d makefile
* | 180be48 only Ifelse now called If
* | ab6682f Merge branch 'decl' of https://github.com/jacobmpenn/Trepp into decl
* | 0b5eaf9 codegen works
* | 565a21c more changes related to semant expressions having types
* | 2e996e9 Merge branch 'decl' of https://github.com/jacobmpenn/Trepp into decl
* | f51bd19 changes to semant
* | b8801f1 added partial nodes
* | ea320d3 only one error
* | 7b270d2 Merge branch 'decl' of https://github.com/jacobmpenn/Trepp into decl
* | 59966fd deleted then
* | e9fab3 Merge branch 'decl' of https://github.com/jacobmpenn/Trepp into decl
* | 6e3a11f Merge branch 'decl' of https://github.com/jacobmpenn/Trepp into decl
* | b269008 Merge branch 'decl' of https://github.com/jacobmpenn/Trepp into decl
* | 1f6fad4 added node parameters into sast
* | dbff0bb less errors
* | eb84547 better but still errors
* | 98ff10f Merge branch 'decl' of https://github.com/jacobmpenn/Trepp into decl
* | 72ebe39 Merge branch 'decl' of https://github.com/jacobmpenn/Trepp into decl
* | 2b979a2 finished adding statement
* | 0edd80f added bind in to vardec
* | a22d8e6 adding a new codegen to play with
* | 4190a81 fixed some bugs
* | 1769214 fixed all changes
* | ea53848 Merge pull request #3 from jacobmpenn/revert-2-jacob_fix
* | 0b675f4 (origin/revert-2-jacob_fix) Revert "added all of jacobs changes"

* | 0b675f4 (origin/revert-2-jacob_fix) Revert "added all of jacobs changes"
* | 1fdad7d Merge pull request #2 from jacobmpenn/jacob_fix
* | 7bf5ece (origin/jacob_fix, jacob_fix) added all of jacobs changes
* | 5cbf5fa fixed merge conflict
* | 79c8e6d Merge pull request #1 from jacobmpenn/codegenwork
* | ef3f23c Merge branch 'decl' into codegenwork
* | a3edc3d changed name of block in codegen to seq, same in sast
* | 4a9d424 bunch of changes: parser: changed funcall to accept actuals list, ast changed to reflect this. same with sast and minor changes.
semant is good through expressions, but nodes have not been added
* | 9412b7f (origin/codegenwork) no duplicate
* | 65461e3 errors
* | 5bf8a41 one error down, new one to fix
* | 66ef710 Merge branch 'decl' of https://github.com/jacobmpenn/Trepp into decl
* | bab6093 added then to prs and scn, incremental semant changes
* | cd734fb parser: fixed typos in ifelse, started changing the semant to reflect our ast
* | 9be89ef first changes to the semant
* | b8541f5 trying to add funcdec
* | 1c721e9 not working but still maybe ok?
* | a09e7af working on swapping node
* | eea91c6 fixed parser
* | b29a74a adding in operations to scanner, fixed waurning for assign
* | 9cb0988 no assign stmt
* | 66513df added nodes into the new structure
* | d4b0e84 func dec better merged with jacob
* | 7ba5f3b minor changes to ast in vdecl and typ
* | 1fdcca8 just warnings in ast and sast
* | 0aeb395 Merge branch 'decl' of https://github.com/jacobmpenn/Trepp into decl
* | e5432a8 new parser reflecting split between statements and fdecls, changes reflectd in parser, ast in progress
* | 8a91d8d fixing some vardec and funcdec errors
* | 6025b0a same
* | 6c03079 string of program additions
* | d8040e7 jacob's verison of ast with functions
* | b0d7ee5 parser
* | 0725c09 fixed some sast
* | b965623 Merge branch 'decl' of https://github.com/jacobmpenn/Trepp into decl

```

```

| \
| * 5548753 fixed some warnings; main error still there
| * 9cb060f Merge branch 'dev' of https://github.com/jacobmpenn/Treapp into dev
| \
| * | 900d3bc got rid of some warnings, still has error
| * | 351238f fixed the semantic bug
| /
| * | cd8c6c0 Merge branch 'dev' of https://github.com/jacobmpenn/Treapp into dev
| \
| \
| * 6528808 added mod and string
| * 34aec39 updated to use Literal (full words)
| * | 8db81d8 changing to sfunc_decl
| /
| * d8e568d added strings to semantic
| * f7d702f Merge branch 'dev' of https://github.com/jacobmpenn/Treapp into dev
| \
| * da7043b added string and string literal
| * 650f1e0 added mod
| * | e724a07 added mod to semantic
| /
| * a24e6e2 small semantic change
| * 1acd985 fixed over arching main
| /
| * 28a3f6e (HEAD -> master, origin/master, origin/HEAD) mircoc
| * 01f1bc2 Initial commit

```



Excluding merges, **4 authors** have pushed **0 commits** to master and **107 commits** to all branches. On master, **0 files** have changed and there have been **0 additions** and **0 deletions**.

