CPS2000 Assignment

Tealang Interpreter

Tony Valentine

Contents

Chars and Arrays

Chars

Support for the `char` type required alterations to the lexer to allow for the recognition of `CharLiterals`, this included a new Single Quote character class as well as new transition states to recognise the `'[0-9]|[A-z]'` pattern. The parser also required the addition of the `char` token when checking for *Type* tokens.

⟨*CharLiteral*⟩ ::= '‘' ⟨*Letter*⟩ | ⟨*Digit*⟩ '’'

⟨*Type*⟩ ::= '`float`' | '`int`' | '`bool`' | '`string`' | '`char`'

⟨*Literal*⟩ ::= ⟨*BooleanLiteral*⟩
 | ⟨*IntegerLiteral*⟩
 | ⟨*FloatLiteral*⟩
 | ⟨*StringLiteral*⟩
 | ⟨*CharLiteral*⟩

Figure 1: EBNF Alterations for Char support

The changes to `SemanticVisitor` and `Interpreter` were also minimal requiring type checking and implementation of operators on `char` types.

Arrays

Array support required little alteration to the lexer outside of including '`[]`' as part of the punctuation character class. On the other hand parsing introduced significantly more issues due to sharing the same first token as *Variable* and *FunctionCall*.

```
⟨Literal⟩ ::= ⟨BooleanLiteral⟩
  |  ⟨IntegerLiteral⟩
  |  ⟨FloatLiteral⟩
  |  ⟨StringLiteral⟩
  |  ⟨ArrayLiteral⟩

⟨ArrayLiteral⟩ ::= '{' ⟨Expression⟩ { ',' ⟨Expression⟩ } '}'

⟨ArrayAccess⟩ ::= ⟨Identifier⟩ '[' ⟨Expression⟩ ']'

⟨Factor⟩ ::= ⟨Literal⟩
  |  ⟨Identifier⟩
  |  ⟨FunctionCall⟩
  |  ⟨SubExpression⟩
  |  ⟨Unary⟩
  |  ⟨ArrayAccess⟩

⟨ArrayAssign⟩ ::= ⟨Identifier⟩ '[' ⟨Expression⟩ ']' '=' ⟨Expression⟩

⟨ArrayDecl⟩ ::= 'let' ⟨Identifier⟩ '[' ⟨Expression⟩ ']' ':' ⟨Type⟩ [ '=' ⟨ArrayLiteral⟩ ]

⟨Statement⟩ ::= ⟨VariableDecl⟩ ';'
  |  ⟨Assignment⟩ ';'
  |  ⟨PrintStatement⟩ ';'
  |  ⟨IfStatement⟩
  |  ⟨ForStatement⟩
  |  ⟨WhileStatement⟩
  |  ⟨RtrnStatement⟩ ';'
  |  ⟨FunctionDecl⟩
  |  ⟨ArrayDecl⟩ ';'
  |  ⟨Block⟩
```

Figure 2: EBNF Alterations for Array Implementation

Parsing

Decleration and Assignment With array assignment sharing the first symbol with variable assignment, and array deceleration sharing two symbols with array deceleration a reworking of the parser was required.

For deceleration statements this involved creating a base `parse_decl` function to read the input stream up to the *Identifier*. From there a lookahead is used to differentiate between array and variable declarations. Array assignments also use a base `parse_assignment` function to determine the correct production with a lookahead.

Expressions In part 1 of the assignment we discussed the process of determining the difference between identifiers and function calls, the process here is nearly identical with introducing a lookahead check for `[` along with the check for `(`.

Semantic Analysis

Array size is declared using an *Expression* type and as such only type checking can be performed during semantic analysis. This applies to both *ArrayDecl* and *ArrayAssign*.

When evaluating *ArrayLiteral* the first elements type is taken as the type for the entire array, if subsequent element types do not agree then an error is thrown. Once the element type checking has completed the `token_type` is set to the corresponding array type equivalent.

The use of unique types for arrays allows for easier checks when performing function calls as a separate flag is not required to differentiate between normal and array types. This was also needed for the implementation of function overloading in question 4.

Interpreter

The use of c++ `any` allows the implementation of arrays to be almost identical to that of normal variables. The `token_type` is used to determine the correct casting before performing any operation.

One important note about the implementation is that the elements in an array are initialised according to c++ vector initialisation. This helps to alleviate memory issues in the implementation when accessing array elements. For an int array of size 5 the initialisation would be; `vector<int>(5)`.

Auto Type Deduction

Changes to the lexer and parser were minimal in this section, `auto` was added as a *Type* in the EBNF and the token was added to the list of keywords in the lexer. The implementation is primarily in the semantic analysis section where upon successful completion produces a properly typed AST.

⟨*Type*⟩ ::= '`float`' | '`int`' | '`bool`' | '`string`' | '`char`' | '`auto`'

Variables

Variable type deduction was implemented by evaluating the type of the expression following the `=` and then editing the AST to change the type from `auto` to the deduced type.

This implementation does break the principle of the Visitor design pattern by altering the structure, however this implementation lead to the cleanest solution.

Functions

Type deduction for functions was more complicated as the type is deduced from *Return* statements, which may contain recursive calls.

The solution involves running the semantic analysis of the *Block* twice, where the first pass tries to set the type of the function from *Return* statements. If no statement returns a concrete type then an error is thrown, otherwise the `ASTFunctionDecl` node is altered and the semantic analysis is run again to ensure type compatibility with possible recursive calls.

Structs

Struct implementation required the most significant amount of overhaul to the parser and the BNF. The most important piece of information is that an *Identifier* can now be used in the same place as a type parameter.

```
⟨StructFunc⟩ ::= ⟨Identifier⟩ '.' ⟨Identifier⟩ '(' ⟨ActualParams⟩ ')'

⟨StructAccess⟩ ::= ⟨Identifier⟩ '.' ⟨Identifier⟩

⟨Factor⟩ ::= ⟨Literal⟩
  |  ⟨Identifier⟩
  |  ⟨FunctionCall⟩
  |  ⟨SubExpression⟩
  |  ⟨Unary⟩
  |  ⟨ArrayAccess⟩
  |  ⟨StructFunc⟩
  |  ⟨StructAccess⟩

⟨StructDecl⟩ ::= 'let' ⟨Identifier⟩ ':' ⟨Identifier⟩ [ '=' ⟨Expression⟩ ]

⟨StructDef⟩ ::= 'tlstruct' ⟨Identifier⟩ '{' { ⟨VariableDecl⟩ ';' | ⟨FunctionDecl⟩ } '}'

⟨FormalParam⟩ ::= ⟨Identifier⟩ ':' ( ⟨Type⟩ | ⟨Identifier⟩ )

⟨FunctionDecl⟩ ::= ( ⟨type⟩ | ⟨Identifier⟩ ) ⟨Identifier⟩ '(' [ ⟨FormalParams⟩ ] ')' ⟨Block⟩

⟨StructAssign⟩ ::= ⟨Identifier⟩ '.' ⟨Identifier⟩ '=' ⟨Expression⟩

⟨Statement⟩ ::= ⟨VariableDecl⟩ ';'
  |  ⟨Assignment⟩ ';'
  |  ⟨PrintStatement⟩ ';'
  |  ⟨IfStatement⟩
  |  ⟨ForStatement⟩
  |  ⟨WhileStatement⟩
  |  ⟨RtrnStatement⟩ ';'
  |  ⟨FunctionDecl⟩
  |  ⟨ArrayDecl⟩ ';'
  |  ⟨StructDecl⟩ ';'
  |  ⟨StructDef⟩
  |  ⟨StructAssign⟩ ';'
  |  ⟨Block⟩
```

Parsing

Statement Parsing

When parsing statements the *Identifier* token can now lead to 4 possible productions

- Variable Assignment
- Array Assignment
- Function Decleration
- Struct Assignment

To deduce the correct production a lookahead token was used, the set of possible tokens for the lookahead is unique and as such can be switched over and the correct production called.

```
case lexer::tok_iden: {
    ll1_tok = lex.getNxtToken();
    // Switching over lookahead token
    switch ((*ll1_tok).type) {
        case lexer::tok_assign: {
            auto x = parse_assignment();
            if (curr_tok.type != lexer::tok_semicolon) {
            fail(";");
            }
            return x;
        }
        case lexer::tok_iden: {
            return parse_function_decl();
        }
        case lexer::tok_decimal: {
            return parse_struct_assign();
        }
        case lexer::tok_square_left:
            return parse_arr_assign();
        }
    break;
}
```

Struct Definition

The struct body is defined to contain either variable or function declarations. The code here can be copied from the updated `parse_statement`, with some extra type checking for `ASTVariableDecl`.

Variable Decleration

Since the `let` token used for *VariableDecl* is also shared with *ArrayDecl* and *StructDecl*, a generic `parse_decl` node was implemented to handle all three deceleration types. Since the return type of this is the base class `ASTStatement`, extra type information needs to be extracted. The implementation leverages c++ `dynamic_cast` which allows for safe casting up and down inheritance hierarchies. The node returned is first cast to `ASTVariableDecl` which is always valid as `ASTStructDecl` and `ASTArrayDecl` are implemented as subclasses. Following this the program attempts to cast the `ASTVariableDecl` node into a `ASTArrayDecl` and `ASTStructDecl` node, if both return null values then the proper type of the node is `ASTVariableDecl`.

```
case lexer::tok_let: {
    auto var = dynamic_cast<ASTVariableDecl *>(parse_decl());
    if (dynamic_cast<ASTArrayDecl *>(var) != NULL) {
      throw std::invalid_argument(
          "Tealang 2 currently only supports var decls in structs");
    }
    if (dynamic_cast<ASTStructDecl *>(var) != NULL) {
      throw std::invalid_argument(
          "Tealang 2 currently only supports var decls in structs");
    }
    node->vars.push_back(var);
    break;
  }

```

Function Decleration

With structs only being defined to contain base variable types the function return types were also limited to supporting base types. Due Tealang currently not supporting `void` return types setter-functions must also include a return statement.

Semantic Analysis

To help with the type deduction and member checking for structs a `Struct` class was created. The class contains two maps, `base_variables` which maps variable names to a `Variable` class (see documentation for Tealang 1) and `base_functions` which maps function names to a `Function` class. The `Struct` class is created and added to the global scope once semantic analysis has been run on the `StructDefn` node in the AST.

```
  class Struct {
  public:
    std::string name;
    std::map<std::string, Variable>
        base_variables; /**< Provides the default initialised variables*/
    std::map<std::string, Function>
        base_functions; /**< Provides all the member functions of a struct */
  };
```

Struct Functions

To check struct functions the type of the variable must first be checked. If the type is `tea_struct` then the `type_name` string can be used to get the corresponding `Struct` class. From there all the function arguments and the function name are passed to the `code_generator` (see part 4) to get the function name. Finally the `base_functions` map can used to check for the existence of the function, and if successful the correct return type is set.

```
void SemanticVisitor::visit(parser::ASTStructFunc *x) {
  Variable var = scope.get_var(x->name);
  if (var.type != parser::tea_struct) {
    throw std::invalid_argument(
        "Invalid operator on variable, it is not a struct in this scope");
  } else {
    std::vector<std::tuple<parser::Tealang_t, std::string>> args;
    for (int i = 0; i < x->args.size(); i++) {
      x->args[i]->accept(this);
      args.push_back({token_type, token_name});
    }
    Struct base_type = scope.get_struct(*var.type_name);
    try {
      Function func =
          base_type.base_functions.at(scope.code_generator(x->element, args));
      token_setter(func.return_type);
    } catch (...) {
      throw std::invalid_argument("Unknown function \"" + x->element +
                                  "\" called for struct:\"" + x->name +
                                  "\" of type \"" + *var.type_name + "\"");
    }
  }
}
```

Struct Access and Assignment

To perform type checking and member checking for structs a `members` map was added to the `Variable` class which links member names to a `Variable`. This approach could have been simplified to just mapping member types, however the code is intended to be re-used by the interpreter.

```cpp
void SemanticVisitor::visit(parser::ASTStructAccess *x) {
  Variable var = scope.get_var(x->name);
  if (var.type != parser::tea_struct) {
    throw std::invalid_argument(
        "Invalid operator on variable, it is not a struct in this scope");
  } else {
    try {
      Variable temp = (*var.members).at(x->element);
      token_setter(temp.type);
    } catch (...) {
      throw std::invalid_argument("Unknown variable \"" + x->element +
                                  "\" called for struct:\"" + x->name +
                                  "\" of type \"" + *var.type_name + "\"");
    }
  }
}
```

Interpreter

The interpreter implementation to support structs is near identical to the semantic analysis. The only difference is that the `member` map is now used as the internal state, and the values of the variables inside are updated during execution.

As mentioned previously during execution variables are stored in a C++ `any`, this allows structs to be returned from functions by only returning the `member` map and setting the `token_type` and `type_name` variables inside of the scope.

```
void Interpreter::visit(parser::ASTStructDecl *x) {
  Variable var;
  var.name = x->identifier;
  var.type_name = x->struct_name;
  var.type = x->Type;
  Struct temp = scope.get_struct(x->struct_name);
  // Decleration with Assignment
  if (x->value) {
    x->value->accept(this);
    // setting member variables
    var.members = std::any_cast<std::map<std::string, Variable>>(token_value);
  } else {
    // using default initialisation
    var.members = temp.base_variables;
  }
  scope.add_var(var);
}
```

Function Overloading

Function overloading is the ability to define functions of the same name, so long as the combination and order of types passed it are unique. Inspired by C and C++ a name mangling scheme was developed to allow for unique code generation for functions.

The implementation works by concatenating the function typenames and the `&` character to the function name. The use of the `&` symbol when concatenating is to prevent a ambiguities when dealing with structs. Given two structs `str` and `ing`, the generated code for `foo(str, ing)` would be `foo&str&ing`, where the `&` symbol clears up the ambiguity with native `string` type.

```cpp
std::string Interpreter::Scope::code_generator( std::string name,
↪  std::vector<std::tuple<parser::Tealang_t, std::string>> args) {
  std::string code = name;
  for (auto arg : args) {
    code += "&" + std::get<1>(arg);
  }
  return code;
}
```

Language Definition

Below is the full Tealang EBNF with changes from TealangV1 and TealangV2 highlighted in blue.

```
⟨Letter⟩ ::= [A-Za-z]

⟨Digit⟩ ::= [0-9]

⟨Printable⟩ ::= [\x20-\x7E]

⟨Type⟩ ::= ‘float’ | ‘int’ | ‘bool’ | ‘string’ | ‘char’ | ‘auto’

⟨BooleanLiteral⟩ ::= ‘true’ | ‘false’

⟨IntegerLiteral⟩ ::= ⟨Digit⟩ { ⟨Digit⟩ }

⟨FloatLiteral⟩ ::= ⟨Digit⟩ { ⟨Digit⟩ } ‘.’ ⟨Digit⟩ { ⟨Digit⟩ }

⟨CharLiteral⟩ ::= ‘‘’ ⟨Letter⟩ | ⟨Digit⟩ ‘’’

⟨StringLiteral⟩ ::= ‘"’ { ⟨Printable⟩ } ‘"’

⟨Literal⟩ ::= ⟨BooleanLiteral⟩
  |  ⟨IntegerLiteral⟩
  |  ⟨FloatLiteral⟩
  |  ⟨StringLiteral⟩
  |  ⟨CharLiteral⟩
  |  ⟨ArrayLiteral⟩

⟨Identifier⟩ ::= ( ‘_’ | ⟨Letter⟩ ) { ‘_’ | ⟨Letter⟩ | ⟨Digit⟩ }

⟨ArrayLiteral⟩ ::= ‘{’ ⟨Expression⟩ { ‘,’ ⟨Expression⟩ } ‘}’

⟨ArrayAccess⟩ ::= ⟨Identifier⟩ ‘[’ ⟨Expression⟩ ‘]’

⟨MultiplicativeOp⟩ ::= ‘*’ | ‘/’ | ‘and’

⟨AdditiveOp⟩ ::= ‘+’ | ‘-’ | ‘or’

⟨RelationalOp⟩ ::= ‘<’ | ‘>’ | ‘==’ | ‘!=’ | ‘<=’ | ‘>=’

⟨ActualParams⟩ ::= ⟨Expression⟩ { ‘,’ ⟨Expression⟩ }

⟨FunctionCall⟩ ::= ⟨Identifier⟩ ‘(’ [ ⟨ActualParams⟩ ] ‘)’

⟨SubExpression⟩ ::= ‘(’ ⟨Expression⟩ ‘)’

⟨Unary⟩ ::= ( ‘-’ | ‘not’ ) ⟨Expression⟩

⟨StructFunc⟩ ::= ⟨Identifier⟩ ‘.’ ⟨Identifier⟩ ‘(’ ⟨ActualParams⟩ ‘)’

⟨StructAccess⟩ ::= ⟨Identifier⟩ ‘.’ ⟨Identifier⟩

⟨Factor⟩ ::= ⟨Literal⟩
  |  ⟨Identifier⟩
  |  ⟨FunctionCall⟩
  |  ⟨SubExpression⟩
  |  ⟨Unary⟩
  |  ⟨ArrayAccess⟩
  |  ⟨StructFunc⟩
  |  ⟨StructAccess⟩

⟨Term⟩ ::= ⟨Factor⟩ { ⟨MultiplicativeOp⟩ ⟨Factor⟩ }

⟨SimpleExpression⟩ ::= ⟨Term⟩ { ⟨AdditiveOp⟩ ⟨Term⟩ }
```

⟨*Expression*⟩ ::= ⟨*SimpleExpression*⟩ { ⟨*RelationalOp*⟩ ⟨*SimpleExpression*⟩ }

⟨*Assignment*⟩ ::= ⟨*Identifier*⟩ ‘`=`’ ⟨*Expression*⟩

⟨*ArrayAssign*⟩ ::= ⟨*Identifier*⟩ ‘`[`’ ⟨*Expression*⟩ ‘`]`’ ‘`=`’ ⟨*Expression*⟩

⟨*VariableDecl*⟩ ::= ‘`let`’ ⟨*Identifier*⟩ ‘`:`’ ⟨*Type*⟩ ‘`=`’ ⟨*Expression*⟩

⟨*ArrayDecl*⟩ ::= ‘`let`’ ⟨*Identifier*⟩ ‘`[`’ ⟨*Expression*⟩ ‘`]`’ ‘`:`’ ⟨*Type*⟩ [‘`=`’ ⟨*ArrayLiteral*⟩]

⟨*StructDecl*⟩ ::= ‘`let`’ ⟨*Identifier*⟩ ‘`:`’ ⟨*Identifier*⟩ [‘`=`’ ⟨*Expression*⟩]

⟨*PrintStatement*⟩ ::= ‘`print`’ ⟨*Expression*⟩

⟨*RtrnStatement*⟩ ::= ‘`return`’ ⟨*Expression*⟩

⟨*IfStatement*⟩ ::= ‘`if`’ ‘(’ ⟨*Expression*⟩ ‘)’ ⟨*Block*⟩ [‘`else`’ ⟨*Block*⟩]

⟨*ForStatement*⟩ ::= ‘`for`’ ‘(’ [⟨*VariableDecl*⟩] ’;’ ⟨*Expression*⟩ ’;’ [⟨*Assignment*⟩] ‘)’ ⟨*Block*⟩

⟨*WhileStatement*⟩ ::= ‘`while`’ ‘(’ ⟨*Expression*⟩ ‘)’ ⟨*Block*⟩

⟨*FormalParam*⟩ ::= ⟨*Identifier*⟩ ‘`:`’ (⟨*Type*⟩ | ⟨*Identifier*⟩)

⟨*FormalParams*⟩ ::= ⟨*FormalParam*⟩ { ‘`,`’ ⟨*FormalParam*⟩ }

⟨*FunctionDecl*⟩ ::= (⟨*type*⟩ | ⟨*Identifier*⟩) ⟨*Identifier*⟩ ‘(’ [⟨*FormalParams*⟩] ‘)’ ⟨*Block*⟩

⟨*StructDef*⟩ ::= ‘`tlstruct`’ ⟨*Identifier*⟩ ‘`{`’ { ⟨*VariableDecl*⟩ ‘`;`’ | ⟨*FunctionDecl*⟩ } ‘`}`’

⟨*StructAssign*⟩ ::= ⟨*Identifier*⟩ ‘`.`’ ⟨*Identifier*⟩ ‘`=`’ ⟨*Expression*⟩

⟨*Statement*⟩ ::= ⟨*VariableDecl*⟩ ‘`;`’
| ⟨*Assignment*⟩ ‘`;`’
| ⟨*PrintStatement*⟩ ‘`;`’
| ⟨*IfStatement*⟩
| ⟨*ForStatement*⟩
| ⟨*WhileStatement*⟩
| ⟨*RtrnStatement*⟩ ‘`;`’
| ⟨*FunctionDecl*⟩
| ⟨*ArrayDecl*⟩ ‘`;`’
| ⟨*StructDecl*⟩ ‘`;`’
| ⟨*StructDef*⟩
| ⟨*StructAssign*⟩ ‘`;`’
| ⟨*Block*⟩

⟨*Block*⟩ ::= ‘`{`’ { ⟨*Statement*⟩ } ‘`}`’

⟨*Program*⟩ ::= { ⟨*Statement*⟩ }

Running Tealang

To compile Tealang navigate to the `src` directory and run `make -B`, the Tealang interpreter will then be compiled into the `tealang` binary.

```
cd src
make -B          # Compiles the program
./tealang --help # Lists the help menu
./tealang <file> # Runs the specified file
```

Tealang Flags

- `-x` or `--xml` will output the generated AST in XML to the specified file
- `-l` or `--lexer` will output the lexer tokens to the specified file
- `-o` or `--output` will redirect `stdout` to the specified file

Testing

Testing Characters

Source code

```
let c1:char = 'h';

let c2:char = 'j';

if(c1==c2){
    print "c1 and c2 are the same";
}else{
    print "c1 and c2 are different";
}

print c1;
print c2;
```

Interpreter output

```
c1 and c2 are different
h
j
```

Generated AST

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root description="Tealang AST Generation">
    <Prog>
        <Decl>
            <Var Type="4">c1</Var>
                <CharConst>h</CharConst>
        </Decl>
        <Decl>
            <Var Type="4">c2</Var>
                <CharConst>j</CharConst>
        </Decl>
        <If>
            <Condition>
                <BinExprNode Op="==">
                    <Identifier>c1</Identifier>
                    <Identifier>c2</Identifier>
                </BinExprNode>
            </Condition>
            <EvalTrue>
                <Block>
                    <Print>
                        <StringConst>c1 and c2 are the same</StringConst>
                    </Print>
                </Block>
            </EvalTrue>
            <Else>
                <Block>
```

```
                    <Print>
                        <StringConst>c1 and c2 are different</StringConst>
                    </Print>
                </Block>
            </Else>
        </If>
        <Print>
            <Identifier>c1</Identifier>
        </Print>
        <Print>
            <Identifier>c2</Identifier>
        </Print>
    </Prog>
</root>
```

Testing Arrays

Source code

```
let x[4]:auto = {1.0,2.0,3.0,4.0};

print x;

x[2+1] = 7.0;

print x[3];

print x;

let z[4]:string = {"hello", "world", "!"};

print z;

let a[2]:char = {'h','i'};

print a;

let b[2]:bool = {true, false};

print b;

let c[2]:int = {10,5};

print c;
```

Interpreter output

```
{1,2,3,4,}
7
{1,2,3,7,}
{hello,world,!,}
{h,i,}
{true,false,}
{10,5,}
```

Generated AST

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root description="Tealang AST Generation">
    <Prog>
        <Decl>
            <Array Type="-1">x</Array>
            <Size>
                <IntConst>4</IntConst>
            </Size>
            <Value>
                <ArrayLiteral>
                    <Item index="0">
                        <FloatConst>1.0</FloatConst>
            </Item>
            <Item index="1">
                <FloatConst>2.0</FloatConst>
            </Item>
            <Item index="2">
                <FloatConst>3.0</FloatConst>
            </Item>
            <Item index="3">
                <FloatConst>4.0</FloatConst>
            </Item>
            </ArrayLiteral>
        </Value>
    </Decl>
    <Print>
        <Identifier>x</Identifier>
    </Print>
    <ArrayAssign>
        <Index>
            <BinExprNode Op="+">
                <IntConst>2</IntConst>
                <IntConst>1</IntConst>
            </BinExprNode>
        </Index>
        <Var>x</Var>
        <FloatConst>7.0</FloatConst>
    </ArrayAssign>
    <Print>
        <ArrayAccess Id="x">
            <Index>
                <IntConst>3</IntConst>
            </Index>
        </ArrayAccess>
    </Print>
    <Print>
        <Identifier>x</Identifier>
    </Print>
    <Decl>
        <Array Type="8">z</Array>
        <Size>
            <IntConst>4</IntConst>
        </Size>
        <Value>
            <ArrayLiteral>
                <Item index="0">
                    <StringConst>hello</StringConst>
                </Item>
                <Item index="1">
                    <StringConst>world</StringConst>
                </Item>
                <Item index="2">
                    <StringConst>!</StringConst>
                </Item>
                </ArrayLiteral>
```

```
                    </Value>
                </Decl>
                <Print>
                    <Identifier>z</Identifier>
                </Print>
                <Decl>
                    <Array Type="9">a</Array>
                    <Size>
                        <IntConst>2</IntConst>
                    </Size>
                    <Value>
                        <ArrayLiteral>
                            <Item index="0">
                                <CharConst>h</CharConst>
                            </Item>
                            <Item index="1">
                                <CharConst>i</CharConst>
                            </Item>
                            </ArrayLiteral>
                        </Value>
                    </Decl>
                    <Print>
                        <Identifier>a</Identifier>
                    </Print>
                    <Decl>
                        <Array Type="7">b</Array>
                        <Size>
                            <IntConst>2</IntConst>
                        </Size>
                        <Value>
                            <ArrayLiteral>
                                <Item index="0">
                                    <BoolConst>true</BoolConst>
                                </Item>
                                <Item index="1">
                                    <BoolConst>false</BoolConst>
                                </Item>
                                </ArrayLiteral>
                            </Value>
                        </Decl>
                        <Print>
                            <Identifier>b</Identifier>
                        </Print>
                        <Decl>
                            <Array Type="6">c</Array>
                            <Size>
                                <IntConst>2</IntConst>
                            </Size>
                            <Value>
                                <ArrayLiteral>
                                    <Item index="0">
                                        <IntConst>10</IntConst>
                                    </Item>
                                    <Item index="1">
                                        <IntConst>5</IntConst>
                                    </Item>
                                    </ArrayLiteral>
                                </Value>
                            </Decl>
                            <Print>
                                <Identifier>c</Identifier>
                            </Print>
                        </Prog>
</root>
```

Testing Auto Type Deduction

Source Code

```
auto fib(a: int){
  if (a < 2) {
    return a;
  }
  return fib(a - 1) + fib(a - 2);
}

auto XGreaterThanY(x:float, y:float){
     return x > y;
}


let x:auto = 2.4;
let y:auto = 2.3;
print XGreaterThanY(x,y);     //true
print fib(12);
```

Interpreter Output

```
true
144
```

Generated AST

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root description="Tealang AST Generation">
    <Prog>
        <FuncDecl>
            <Var Type="-1">fib</Var>
            <Arg typename="int">a</Arg>
            <Block>
                <If>
                    <Condition>
                        <BinExprNode Op="&lt;">
                            <Identifier>a</Identifier>
                            <IntConst>2</IntConst>
                        </BinExprNode>
                    </Condition>
                    <EvalTrue>
                        <Block>
                            <Return>
                                <Identifier>a</Identifier>
                            </Return>
                        </Block>
                    </EvalTrue>
                    <Else>
                    </Else>
                </If>
                <Return>
                    <BinExprNode Op="+">
```

```
                        <FuncCall Id="fib">
                            <BinExprNode Op="-">
                                <Identifier>a</Identifier>
                                <IntConst>1</IntConst>
                            </BinExprNode>
                        </FuncCall>
                        <FuncCall Id="fib">
                            <BinExprNode Op="-">
                                <Identifier>a</Identifier>
                                <IntConst>2</IntConst>
                            </BinExprNode>
                        </FuncCall>
                    </BinExprNode>
                </Return>
            </Block>
        </FuncDecl>
        <FuncDecl>
            <Var Type="-1">XGreaterThanY</Var>
            <Arg typename="float">x</Arg>
            <Arg typename="float">y</Arg>
            <Block>
                <Return>
                    <BinExprNode Op="&gt;">
                        <Identifier>x</Identifier>
                        <Identifier>y</Identifier>
                    </BinExprNode>
                </Return>
            </Block>
        </FuncDecl>
        <Decl>
            <Var Type="-1">x</Var>
                <FloatConst>2.4</FloatConst>
        </Decl>
        <Decl>
            <Var Type="-1">y</Var>
                <FloatConst>2.3</FloatConst>
        </Decl>
        <Print>
            <FuncCall Id="XGreaterThanY">
                <Identifier>x</Identifier>
                <Identifier>y</Identifier>
            </FuncCall>
        </Print>
        <Print>
            <FuncCall Id="fib">
                <IntConst>12</IntConst>
            </FuncCall>
        </Print>
    </Prog>
</root>
```

Testing Structs

Source Code

```
tlstruct Vector {
    let x:float = 0.0;
    let y:float = 0.0;
    let z:float = 0.0;

    int scale(s:float){
        x = x * s;
        y = y * s;
        z = z * s;
        return 0;
    }

    int translate(tx:float, ty:float, tz:float){
        x = x + tx;
        y = y + tz;
        z = z + tz;
        return 0;
    }

}

auto Add(v1:Vector, v2:Vector){
    let vTemp:Vector;
    vTemp.x = v1.x + v2.x;
    vTemp.y = v1.y + v2.y;
    vTemp.z = v1.z + v2.z;
    return vTemp;
}

let v1:Vector;
v1.x = 1.0;
v1.y = 2.0;
v1.z = 2.0;

let v2:Vector;
v2.x = 2.0;
v2.y = 1.2;
v2.z = 0.0;

let v3:Vector = Add(v1,v2);
print v3.x; // 3
print v3.y; // 3.2
print v3.z; // 2

let tempVar:int = v3.translate(1.0,1.0,1.0); // adds 1 to every element

print v3;

let v4:Vector = Add(v1,v3);

```

```
print v4;

tempVar = v1.scale(10.0);

print v1;
```

Interpreter Output

```
3
3.2
2
Vector:{
   x = 4,
   y = 4.2,
   z = 3,
}
Vector:{
   x = 5,
   y = 6.2,
   z = 5,
}
Vector:{
   x = 10,
   y = 20,
   z = 20,
}
```

Generated AST

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root description="Tealang AST Generation">
    <Prog>
        <StructDefn name="Vector">
            <Variables>
                <Decl>
                    <Var Type="0">x</Var>
                        <FloatConst>0.0</FloatConst>
                </Decl>
                <Decl>
                    <Var Type="0">y</Var>
                        <FloatConst>0.0</FloatConst>
                </Decl>
                <Decl>
                    <Var Type="0">z</Var>
                        <FloatConst>0.0</FloatConst>
                </Decl>
            </Variables>
            <Functions>
                <FuncDecl>
                    <Var Type="1">scale</Var>
                    <Arg typename="float">s</Arg>
                    <Block>
                        <Assign>
                            <Var>x</Var>
                            <BinExprNode Op="*">
                                <Identifier>x</Identifier>
                                <Identifier>s</Identifier>
                            </BinExprNode>
                        </Assign>
                        <Assign>
                            <Var>y</Var>
                            <BinExprNode Op="*">
                                <Identifier>y</Identifier>
                                <Identifier>s</Identifier>
                            </BinExprNode>
                        </Assign>
                        <Assign>
                            <Var>z</Var>
                            <BinExprNode Op="*">
                                <Identifier>z</Identifier>
                                <Identifier>s</Identifier>
                            </BinExprNode>
                        </Assign>
                        <Return>
                            <IntConst>0</IntConst>
                        </Return>
                    </Block>
                </FuncDecl>
                <FuncDecl>
                    <Var Type="1">translate</Var>
                    <Arg typename="float">tx</Arg>
                    <Arg typename="float">ty</Arg>
                    <Arg typename="float">tz</Arg>
                    <Block>
                        <Assign>
                            <Var>x</Var>
                            <BinExprNode Op="+">
                                <Identifier>x</Identifier>
                                <Identifier>tx</Identifier>
                            </BinExprNode>
                        </Assign>
                        <Assign>
                            <Var>y</Var>
                            <BinExprNode Op="+">
                                <Identifier>y</Identifier>
                                <Identifier>tz</Identifier>
                            </BinExprNode>
                        </Assign>
                        <Assign>
                            <Var>z</Var>
                            <BinExprNode Op="+">
                                <Identifier>z</Identifier>
                                <Identifier>tz</Identifier>
                            </BinExprNode>
                        </Assign>
                        <Return>
```

```
                    <IntConst>0</IntConst>
                </Return>
            </Block>
        </FuncDecl>
    </Functions>
</StructDefn>
<FuncDecl>
    <Var Type="-1">Add</Var>
    <Arg typename="Vector">v1</Arg>
    <Arg typename="Vector">v2</Arg>
    <Block>
        <Decl>
            <Var Type="Vector">vTemp</Var>
        </Decl>
        <StructAssign name="vTemp" element="x">
            <BinExprNode Op="+">
                <StructAccess name="v1" element="x"/>
                <StructAccess name="v2" element="x"/>
            </BinExprNode>
        </StructAssign>
        <StructAssign name="vTemp" element="y">
            <BinExprNode Op="+">
                <StructAccess name="v1" element="y"/>
                <StructAccess name="v2" element="y"/>
            </BinExprNode>
        </StructAssign>
        <StructAssign name="vTemp" element="z">
            <BinExprNode Op="+">
                <StructAccess name="v1" element="z"/>
                <StructAccess name="v2" element="z"/>
            </BinExprNode>
        </StructAssign>
        <Return>
            <Identifier>vTemp</Identifier>
        </Return>
    </Block>
</FuncDecl>
<Decl>
    <Var Type="Vector">v1</Var>
</Decl>
<StructAssign name="v1" element="x">
    <FloatConst>1.0</FloatConst>
</StructAssign>
<StructAssign name="v1" element="y">
    <FloatConst>2.0</FloatConst>
</StructAssign>
<StructAssign name="v1" element="z">
    <FloatConst>2.0</FloatConst>
</StructAssign>
<Decl>
    <Var Type="Vector">v2</Var>
</Decl>
<StructAssign name="v2" element="x">
```

```
            <FloatConst>2.0</FloatConst>
        </StructAssign>
        <StructAssign name="v2" element="y">
            <FloatConst>1.2</FloatConst>
        </StructAssign>
        <StructAssign name="v2" element="z">
            <FloatConst>0.0</FloatConst>
        </StructAssign>
        <Decl>
            <Var Type="Vector">v3</Var>
                <FuncCall Id="Add">
                    <Identifier>v1</Identifier>
                    <Identifier>v2</Identifier>
                </FuncCall>
        </Decl>
        <Print>
            <StructAccess name="v3" element="x"/>
        </Print>
        <Print>
            <StructAccess name="v3" element="y"/>
        </Print>
        <Print>
            <StructAccess name="v3" element="z"/>
        </Print>
        <Decl>
            <Var Type="1">tempVar</Var>
                <StructFunc name="v3" element="translate">
                    <FloatConst>1.0</FloatConst>
                    <FloatConst>1.0</FloatConst>
                    <FloatConst>1.0</FloatConst>
                </StructFunc>
        </Decl>
        <Print>
            <Identifier>v3</Identifier>
        </Print>
        <Decl>
            <Var Type="Vector">v4</Var>
                <FuncCall Id="Add">
                    <Identifier>v1</Identifier>
                    <Identifier>v3</Identifier>
                </FuncCall>
        </Decl>
        <Print>
            <Identifier>v4</Identifier>
        </Print>
        <Assign>
            <Var>tempVar</Var>
            <StructFunc name="v1" element="scale">
                <FloatConst>10.0</FloatConst>
            </StructFunc>
        </Assign>
        <Print>
            <Identifier>v1</Identifier>
        </Print>
    </Prog>
</root>
```

Testing Function Overloading

Source Code

```
float AverageOfThree (x:float, y:float, z:float){
    print "Float Average of three";
    let total:float = x+y+z;
    return total/3.0;
}

int AverageOfThree (x:int, y:int, z:int){
    print "Integer Average of three";
    let total:int = (x+y+z)/3;
    return total;
}

print AverageOfThree(10,20,45);

print AverageOfThree(12.5,7.5,30.5);
```

Interpreter Output

```
Integer Average of three
25
Float Average of three
16.8333
```

Generated AST

```
<?xml version="1.0" encoding="UTF-8"?>
<root description="Tealang AST Generation">
    <Prog>
        <FuncDecl>
            <Var Type="0">AverageOfThree</Var>
            <Arg typename="float">x</Arg>
            <Arg typename="float">y</Arg>
            <Arg typename="float">z</Arg>
            <Block>
                <Print>
                    <StringConst>Float Average of three</StringConst>
                </Print>
                <Decl>
                    <Var Type="0">total</Var>
                        <BinExprNode Op="+">
                            <Identifier>x</Identifier>
                            <BinExprNode Op="+">
                                <Identifier>y</Identifier>
                                <Identifier>z</Identifier>
                            </BinExprNode>
                        </BinExprNode>
                </Decl>
                <Return>
                    <BinExprNode Op="/">
                        <Identifier>total</Identifier>
                        <FloatConst>3.0</FloatConst>
                    </BinExprNode>
                </Return>
            </Block>
        </FuncDecl>
        <FuncDecl>
            <Var Type="1">AverageOfThree</Var>
            <Arg typename="int">x</Arg>
            <Arg typename="int">y</Arg>
            <Arg typename="int">z</Arg>
            <Block>
                <Print>
                    <StringConst>Integer Average of three</StringConst>
                </Print>
                <Decl>
                    <Var Type="1">total</Var>
                        <BinExprNode Op="/">
                            <BinExprNode Op="+">
                                <Identifier>x</Identifier>
                                <BinExprNode Op="+">
                                    <Identifier>y</Identifier>
                                    <Identifier>z</Identifier>
                                </BinExprNode>
                            </BinExprNode>
                            <IntConst>3</IntConst>
                        </BinExprNode>
                </Decl>
                <Return>
                    <Identifier>total</Identifier>
                </Return>
            </Block>
        </FuncDecl>
        <Print>
            <FuncCall Id="AverageOfThree">
                <IntConst>10</IntConst>
                <IntConst>20</IntConst>
                <IntConst>45</IntConst>
            </FuncCall>
        </Print>
        <Print>
            <FuncCall Id="AverageOfThree">
                <FloatConst>12.5</FloatConst>
                <FloatConst>7.5</FloatConst>
                <FloatConst>30.5</FloatConst>
            </FuncCall>
        </Print>
    </Prog>
</root>
```

References