Semantic Analysis

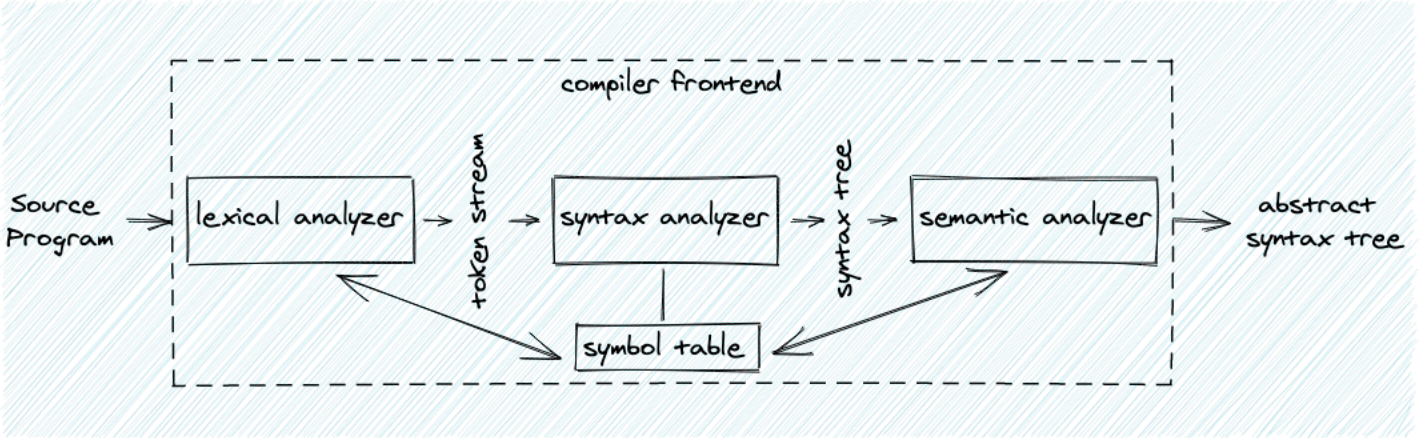
Suejb Memeti

January 2022

Outline

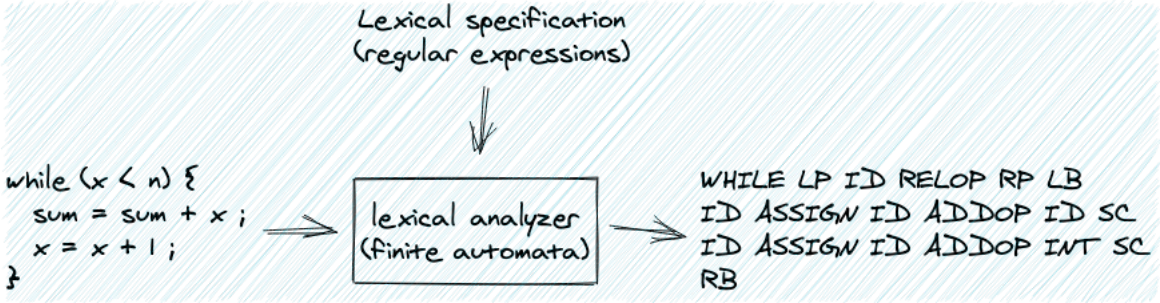
- Quick overview
- Tree traversal
  - Base-class tree traversal
  - OO design
- Symbol Tables
  - Scope
  - Implementation details
- Type checking
  - Implementation details

Frontend Overview



- Lexical analysis
  - Identify atomic language constructs
  - Ex: if -> IF; 2.0 -> FLOAT; x -> ID; …
- Syntax analysis
  - Check if the token stream is correct with respect to language specification
- Semantic analysis
  - Check type relations and consistency rules
  - E.g. if type(lhs) = type(rhs) in an assignment lhs=rhs; …
- Symbol table
  - Contains information about the source program. Used by all phases of the compiler.

Lexical Analysis Overview

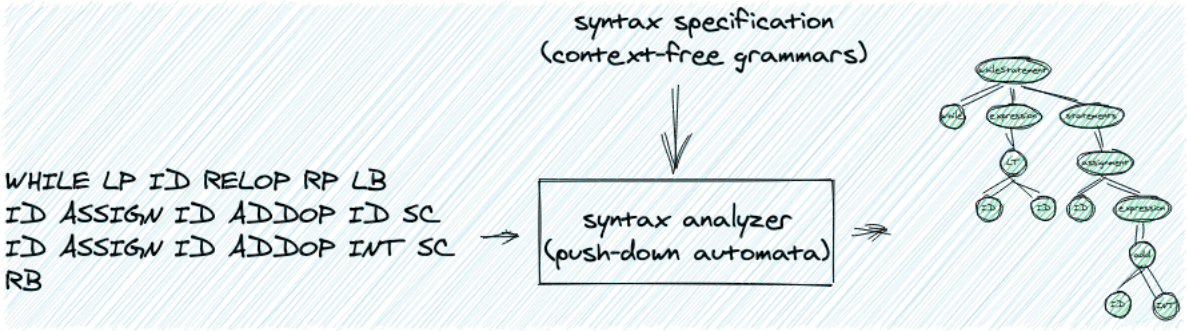


- Input: program representations (character sequence)
- Output: program representation (token sequence)

- A token comprises: type, literal value, location information

- Lexical specification: regular expressions
- Lexical analysis implementation: finite automata

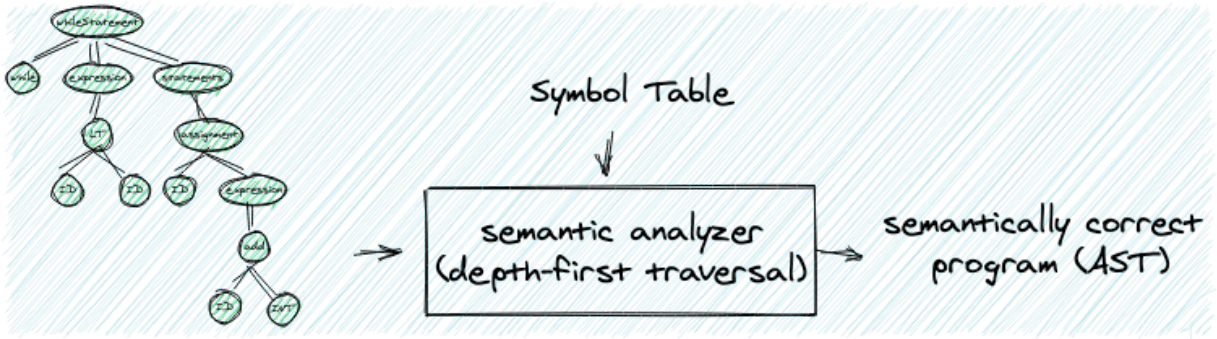
Syntax Analysis (Parsing) Overview



- Input: program representation (token sequence)
- Output: parse (syntax) tree
- A parse tree comprises nodes and non-labeled edges.

- Syntax specification: context free grammar
- Syntax analysis implementation: push-down automata
  - Top-down and bottom-up parsers

Semantic Analysis (Type checking) Overview

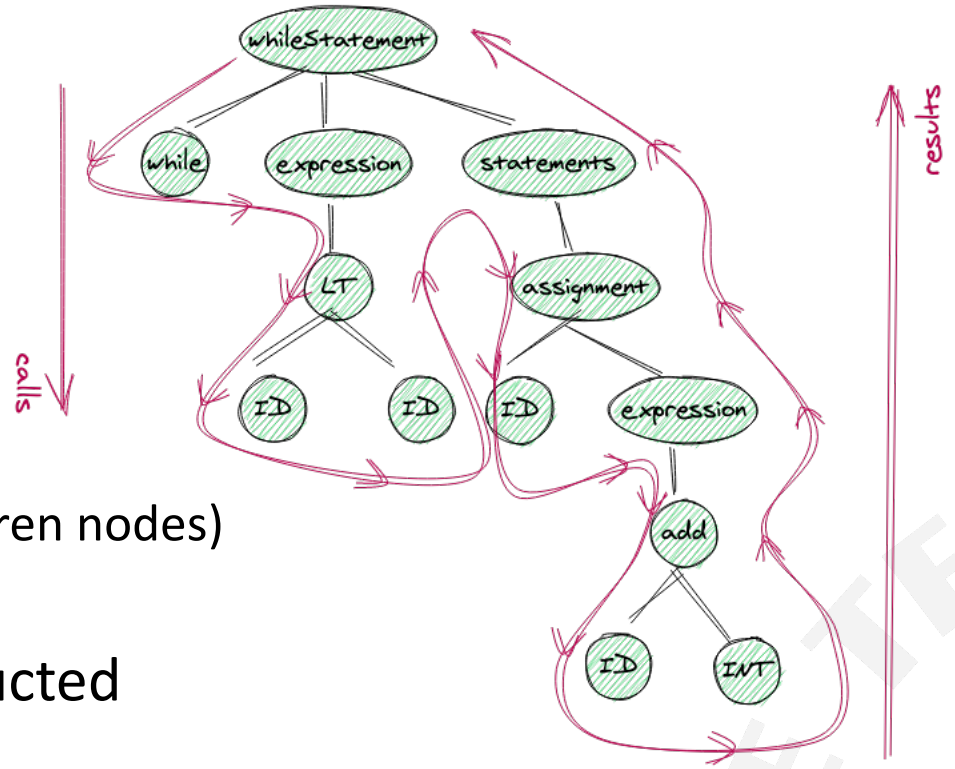


- Input: abstract syntax tree
- Output: semantically correct program

- Symbol table: constructed by visiting the AST
- Semantic analysis: performed by traversing the AST and using the information in the symbol table

Tree traversal

- Tree traversal
  - Recursive in nature
  - Pre-order
  - Post-order

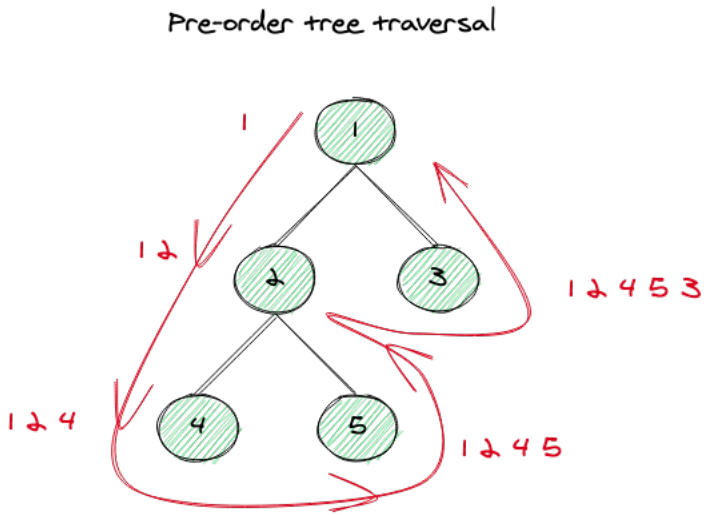


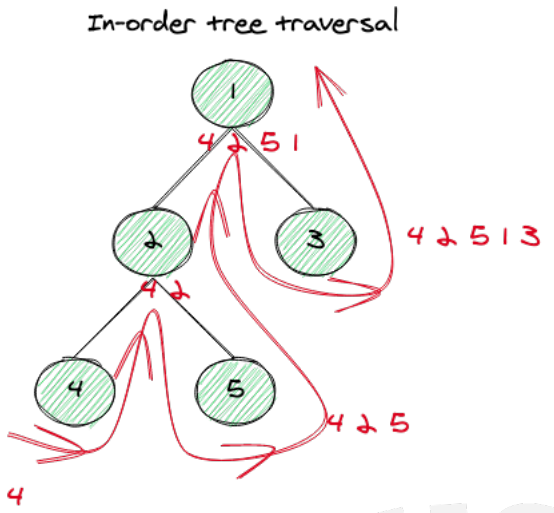
- E.g., the print_tree() method
  - Pre-order (starts from root visit all children nodes)

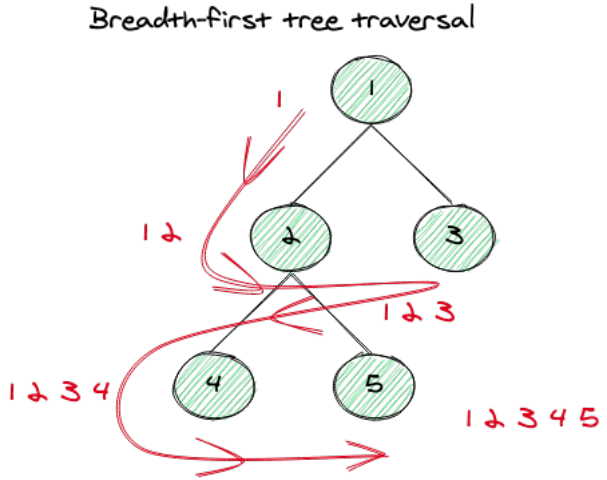
- Depending on how the AST is constructed
  - Base design
    - Determine what actions to perform based on the type node
  - OO-design
    - A base class Node is derived by specific classes that represent specific language constructs. Each derived class implements its own functionality.

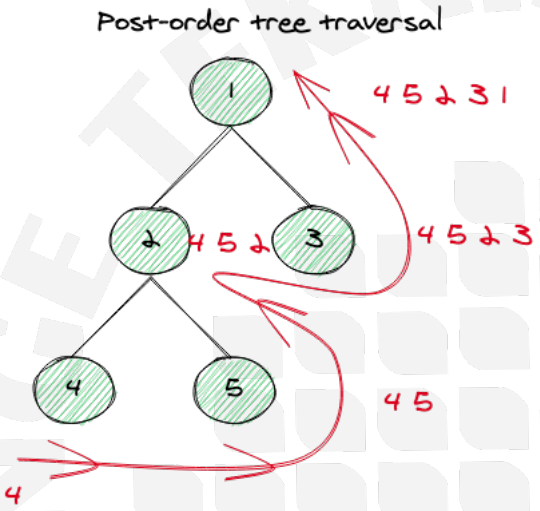
Tree traversal algorithms (overview)

- **Depth-first traversal**
  - In-order (Left, Root, Right)
    - E.g., 4 2 5 1 3
  - **Pre-order** (Root, Left, Right)
    - E.g., 1 2 4 5 3
    - Used to construct the symbol table
  - **Post-order** (Left, Right, Root)
    - E.g., 4 5 2 3 1
    - Used to perform the semantic analysis (type checking)
- Breadth-first traversal
  - E.g., 1 2 3 4 5





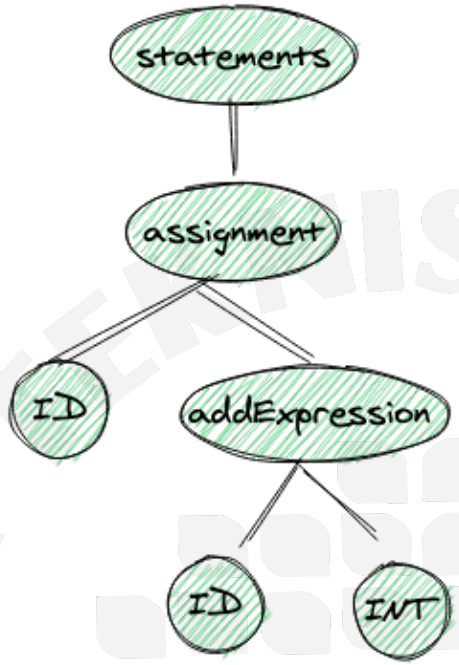




Tree traversal – base design

- A single class
- The execute method is called recursively
- Depending on what type the Node is
  - Perform specific actions

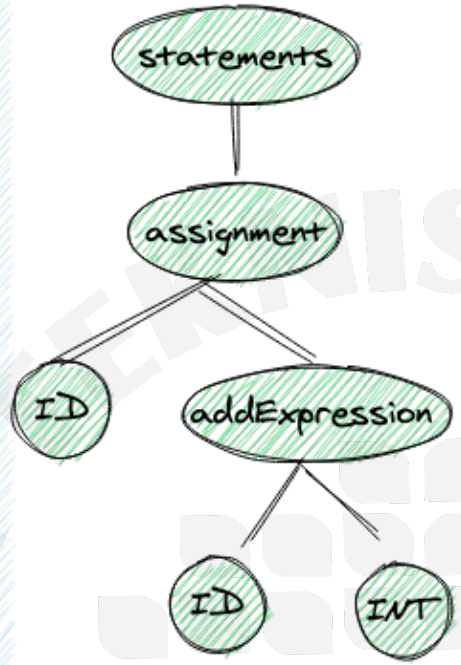
```
execute(...) {
  ...
  if(type == "statements") {
     visit all statements
  }
  else if(type == "assignment") {
     lhs = execute(child[0]);
     rhs = execute(child[1]);
  }
  ...
  else if(type == "addExpression") {
     lhs = execute(child[0]);
     rhs = execute(child[1]);
     result = lhs + rhs;
  }
...
}
```



Tree traversal – OO design

- A base class
  - A set of derived classes
- The execute method is called recursively
- Each derived class has its own implementation of the execute method

```
Node() {
  ...
  execute(...) {...}
  ...
}
Statements :: Node() {
  ...
  execute(...) {
    visit all statements
  }
  ...
}
Assignment :: Node() {
  ...
  execute(...) {
    lhs = execute(child[0]);
    rhs = execute(child[1]);
    ...
  }
  ...
}
AddExpression :: Node() {
  ...
  execute(...) {
    lhs = execute(child[0]);
    rhs = execute(child[1]);
    result = lhs + rhs;
  }
  ...
}
```



Symbol table

- Data structures to hold information about the program constructs
  - E.g., variable/method/object/class… name, type, and scope

- Can be constructed during lexical analysis and syntax analysis; or after
- Used during semantic analysis, code generation, optimization, error detection

- Can be constructed in one (pre-order) traversal of the AST
- Each identifier is represented by a record where information is stored

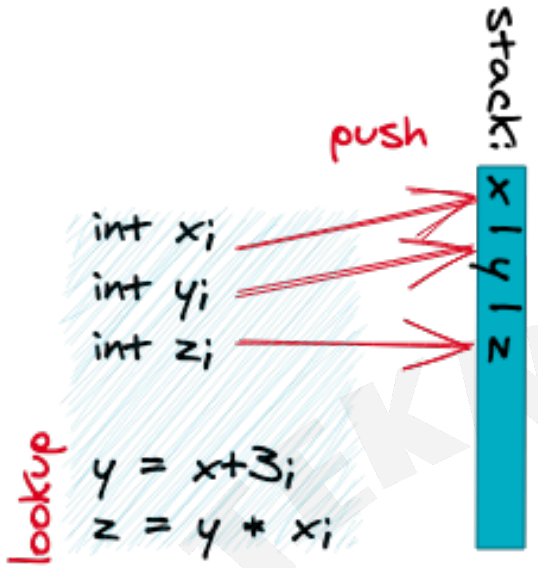
- Features: lookup, insert, modify, delete

- Symbol tables need to support multiple declarations of the same identifier within a program

- Common implementations: trees, linked lists, hash tables

A simple symbol table

- Implemented as a stack
- Supports variable declarations
  - Symbols are added one at a time
- Operations
  - add_symbol(x) – push x to stack
  - find_symbol(x) – search stack for x. return null if not found.
  - remove_symbol(x) – pop the stack
- Does not support
  - Method declarations, method parameters,
  - Variable declarations within methods



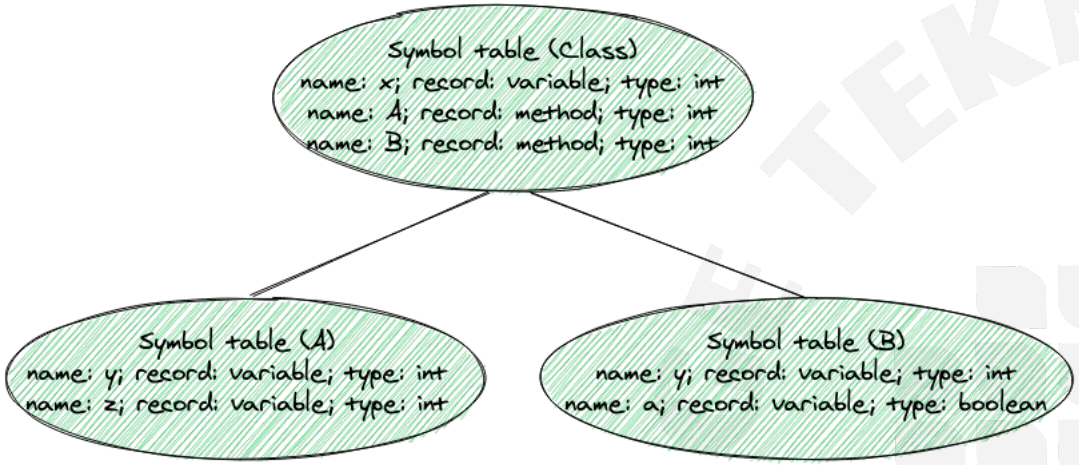
A more advanced symbol table

```
int x;
int A() {
  int y;
  int z;
}
int B() {
  int y;
  boolean true;
}
```

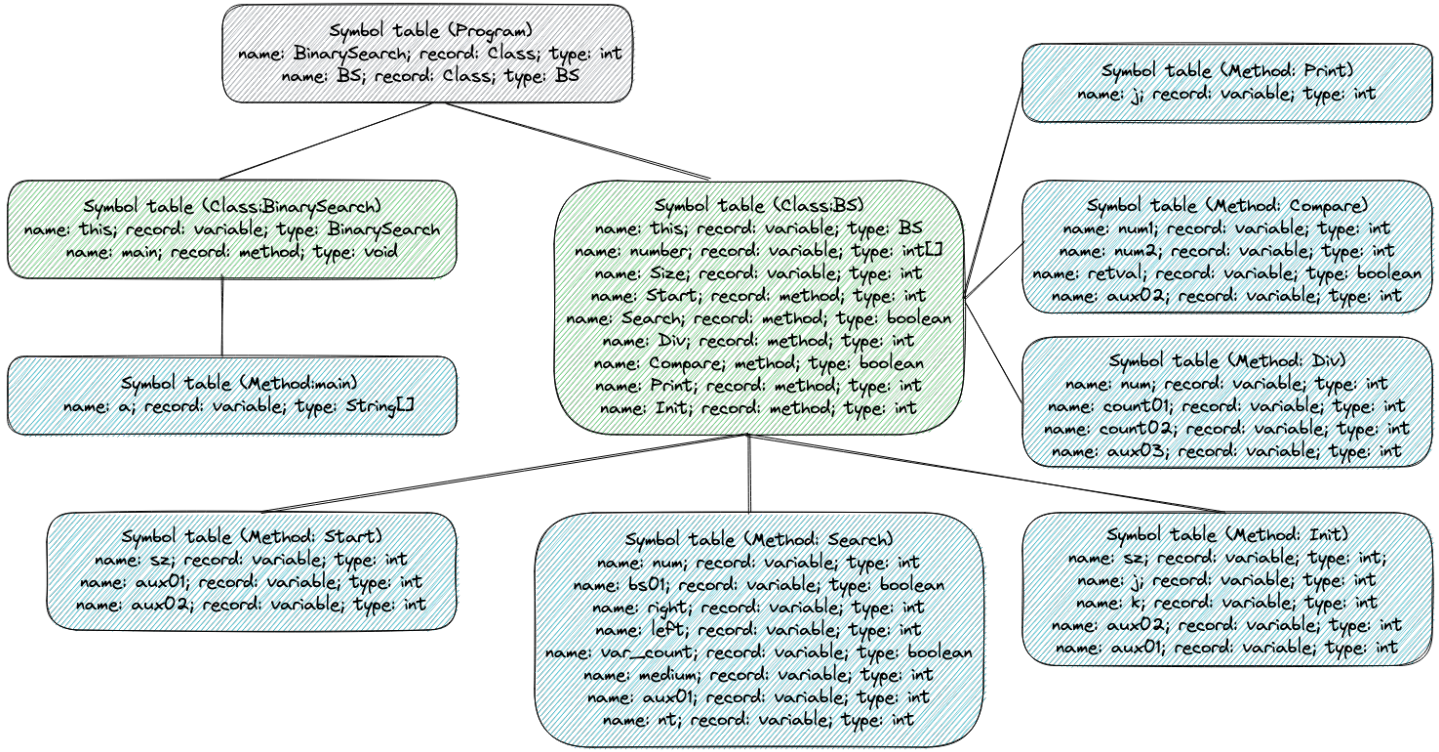
- Implemented as a stack
- Supports variable declarations, method, parameters, …

- Operations
  - enter_scope() – start/push a new nested scope
  - exit_scope() – exists/pops the current scope
  - add_symbol(x) – push x to stack
  - find_symbol(x) – search stack for x. return null if not found.
  - check_scope(x) – returns true if x is defined in current scope
  - remove_symbol(x) – pop the stack

- Function names can be used before their definitions?
  - Pass 1: construct the symbol table
  - Pass 2: perform the semantic analysis
    - Semantic analysis may require more than one pass



Symbol Table: example




```
class BinarySearch {
    public static void main(String[] a) {
        System.out.println(new BS().Start(20));
    }
}
class BS {
    int[] number;
    int size;

    public int Start(int sz) {
        int aux01;
        int aux02;
        aux01 = this.Init(sz);
        aux02 = this.Print();
        if (this.Search(8)) System.out.println(1);
        else System.out.println(0);
        ...
        return 999;
    }

    public boolean Search(int num) {
        boolean bs01;
        int right;
        int left;
        boolean var_cont;
        int medium;
        int aux01;
        int nt;

        aux01 = 0;
        bs01 = false;
        ...
        return bs01;
    }
```

```
    public int Div(int num) {
        int count01;
        int count02;
        int aux03;

        count01 = 0;
        ...
        return count01;
    }

    public boolean Compare(int num1, int num2) {
        boolean retval;
        int aux02;

        retval = false;
        ...
        return retval;
    }

    public int Print() {
        int j;

        j = 1;
        while (j < (size)) {
            System.out.println(number[j]);
        ...
        return 0;
    }

    public int Init(int sz) {
        int j;
        int k;
        int aux02;
        int aux01;

        size = sz;
        ...
        return 0;
    }
}
```

Tree-based symbol table implementation

```
class Record {
    string id;
    string type;
    //getters and setters
    printRecord();
}

class Class: Record {
   map<string, Variable> variables;
   map<string, Method> methods;

   addVariable(){...}
   addMethod(){...}
   lookupVariable(){...}
   lookupMethod(){...}
   ...
}

class Method: Record {
   map<string, Variable> parameters;
   map<string, Variable> variables;

   addVariable(){...}
   addParameter(){...}
   ...
}

class Variable: Record {
   ...
}
```

```
class SymbolTable {
   Scope root;
   Scope current;

   SymbolTable(){root = new Scope(null); current = root;}
   enterScope(){current = current.nextChild(); //create new scope if needed}
   exitScope(){current = current.Parent();}

   put(String key, Record item) {current.put(key, item);}
   Record lookup(String key) {return current.lookup(key);}

   printTable() {root.printScope();}
   resetTable() {root.resetScope(); //preparation for new traversal}
}

class Scope{
   int next = 0; //next child to visit
   Scope parentScope; //parent scope
   list childrenScopes; //children scopes
   map records; //symbol to record map
   ...

   Scope nextChild() {
     Scope nextChild;
     if(next == childrenScopes.size()) { //create new child Scope
         nextChild = new Scope(this);
         childrenScopes.add(nextChild);
     } else { nextChild = childrenScopes.get(next); } //visit scope
     next++;
     return nextChild;
   }

   ...
}
```

```
class Scope{
   ...

   Record lookup(string key) {
     if(records.containsKey(key)) // does it exist in the current scope?
       return record.get(key);
     else {
       if(parent == null)
          return null; //identifier not in the symbol table
       else
          return parent.lookup(key); // delegate the request to parent scope
     }
   }

   resetScope(){
     next = 0;
     for(int i = 0; i < children.size(); i++)
        children.get(i).resetScope();
   }
}
```

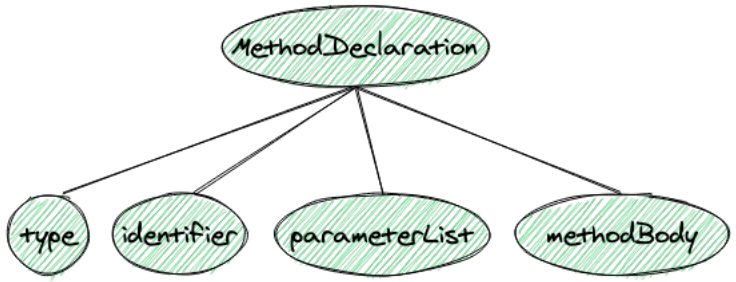
Symbol table summary

- Initialize the symbol table object
- Traverse the tree left-to-right
- For each detected scope in the tree
  - Create a new scope node in the symbol table
  - Call symbtolTable.enterScope()
- New identifiers detected
  - Classes, methods, fields, parameters, variables
  - Create a new Record with id and type
  - Add the Record to the current scope
    - symbolTable.put(id, new Record(id, type))
  - Leaving a scope
    - sybmolTable.exitScope()

- Focus on identifier declarations; their use and definitions is for later
- Single AST traversal is enough
  - Implement a tree traversal

- Where do we shift scope?
- In which AST nodes do we put new records in the table?

Generating the ST - example




```
Node() {
  ...
  execute(...) {...}
  ...
}
MethodDeclaration :: Node() {
  ...
  execute(...) {
    retType = execute(child[0]); //return type
    name = execute(child[1]); //method name
    currentMethod = new Method(mName, retType);
    symbolTable.put(mName, currentMethod); //add to ST
    currentClass.addMethod(mName, currentMethod); //add to current class

    symbolTable.enterScope(); //enter method scope
    execute(child[2]); //visist parameterList
    execute(child[3]); //visit methodBody
    symbolTable.exitScope(); //exit scope
    ...
  }
  ...
}
```

Semantic analysis

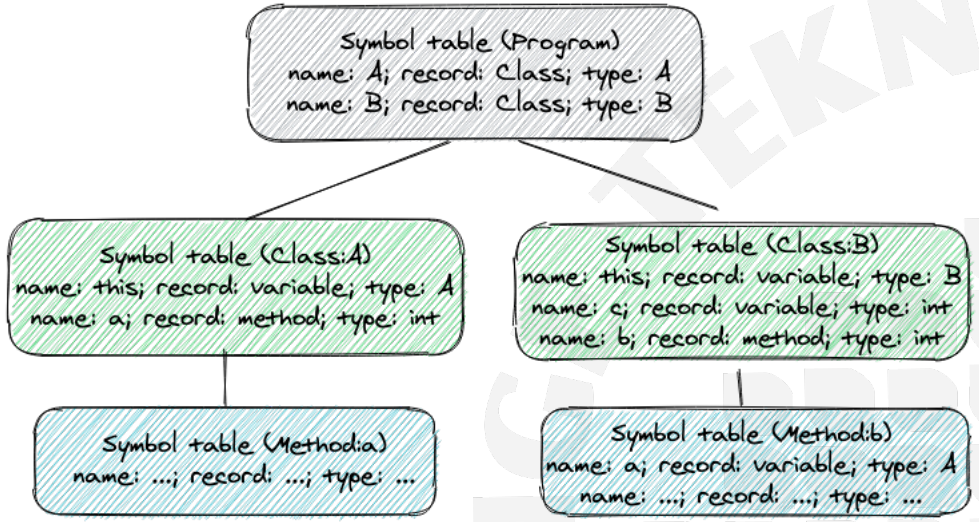
- A separate phase of the compiler
  - Input: AST and symbol table
  - Implemented as a new tree traversal (may require more than one tree traversal)
  - Requires evaluation of types of subtrees (in post-order)
- Tasks
  - Verify that all identifiers are declared
  - Type checking
    - Expressions: e.g., In **a+b;** both a and b should be integers
    - Statements: e.g., In **a = b;** type of a and b should be the same
    - Methods: In **int a(int c){… return b;}** the type of return expression (b) should be the same as type of the method declaration (int).
    - Method calls: In **a(x);** the type of x and c should be the same
    - Array access: In **b[a]; b.length** the type of a must be integer; the type of b must be int[];
- Out of scope
  - Verify that variables have a value before they are used
  - Inheritance, polymorphic calls, and ither OO features

Undeclared identifiers

```
class A {
  int a(){...}
}

class B{
  int c;
  int b(){
    A a;
    ...
    res = a.a() + c;
  }
}
```

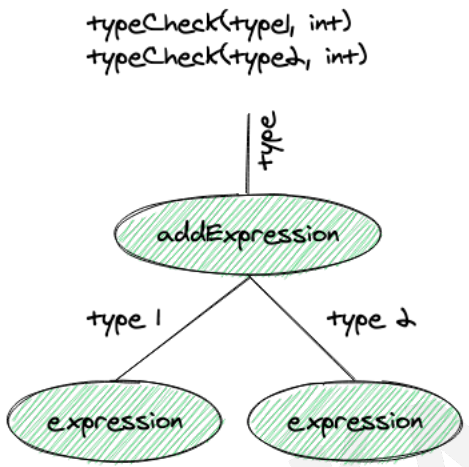
- All declared identifiers are stored in the symbol table
  - Lookup: symbolTable.lookup(c) should not be null
- Dealing with method calls
  - Method a() is not visible from b()
    - Simple lookup for a() will return null
  - Solution
    - Lookup(a).getType() will result with A;
    - Lookup(A).getMethod(a) will not be null;

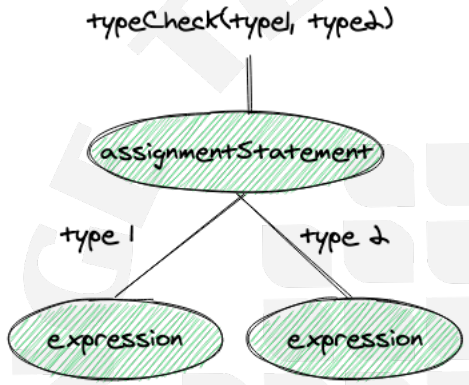


Type checking

- Type information is transported upwards
  - The evaluate method returns the type of a child
  - Leaf nodes return their types
  - Other nodes need to evaluate the type of their children
- When the type of all children nodes is evaluated, we perform type checking

```
Node() {
  ...
  semanticAnalysis(...) {...}
  ...
}
AddExpression :: Node() {
  ...
  semanticAnalysis(...) {
    ...
    for(Node i : children) {
      rhsType = evaluate(children[i]);
      if(lhsType != "int")
        print("Type missmatch");
    ...
  }
...
AssignmentStatement :: Node() {
  ...
  semanticAnalysis(...) {
    ...
    lhsType = evaluate(child[0]);
    rhsType = evaluate(child[1]);
    if(lhsType != rhsType)
       print("Type missmatch");
    ...
  }
}
```

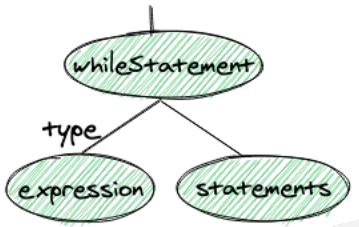


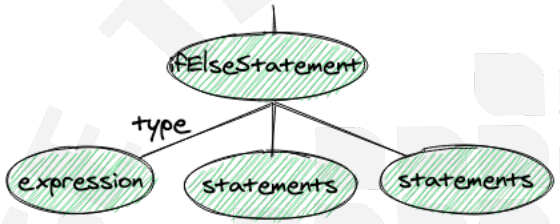


If/While Statements

- We need to evaluate the type of the boolean expression
- The statements AST nodes do not have any types
- Other AST nodes (e.g., expressions) have types

```
Node() {
  ...
  semanticAnalysis(...) {...}
  ...
}
WhileStatement :: Node() {
  ...
  semanticAnalysis(...) {
    ...
    conditionType = evaluate(child[0]);
    if(conditionType != "boolean")
       print("Type missmatch");
    ...
  }
...
IfExpression :: Node() {
  ...
  semanticAnalysis(...) {
    ...
    conditionType = evaluate(child[0]);
    if(conditionType != "boolean")
       print("Type missmatch");
    ...
  }
}
```

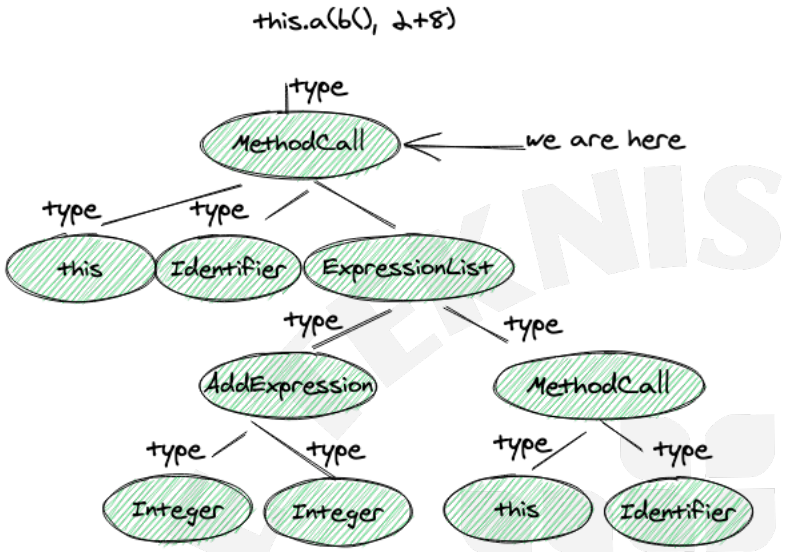




Method calls

- Get the target (this) type class
  - Lookup if class exists in ST
- Get method name
  - Lookup method in the scope of the target class
  - Check if method return type is same as declared type
- Evaluate all parameters
  - Check if the type of parameters corresponds to the declared method parameters

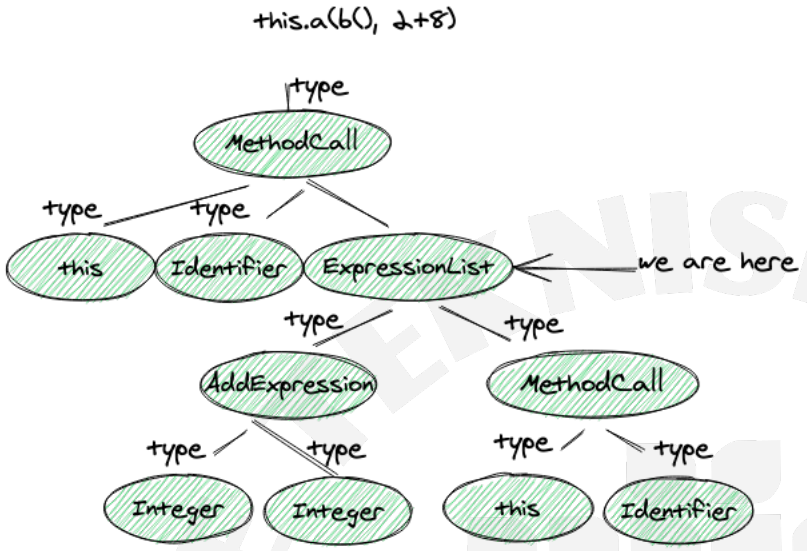
```
Node() {
  ...
  semanticAnalysis(...) {...}
  ...
}
MethodCall :: Node() {
  ...
  semanticAnalysis(...) {
    ...
    targetType = evaluate(child[0]);
    class = symbolTable.lookup(targetType);
    methodName = evaluate(child[1]);
    method = class.getMethod(methodName);
    ...
    evaluate(child[3]);
    ...
  }
}
```



Parameter List

- Check the size of defined list of parameters and children
- For each child
  - Check if type is the same as the defined parameter type
- You need a reference to the current method

```
Node() {
  ...
  semanticAnalysis(...) {...}
  ...
}
ExpressionList :: Node() {
  ...
  semanticAnalysis(...) {
    ...
    paramList = currentMethod.children.size();
    if(paramListCount != children.size())
        print("parameter list size mismatch");
      for(int i = 0; i < children.size(); i++) {
        type = evaluate(children[i]);
        definedType = evaluate(paramList.children[i]);
        if(type != definedType)
          print(parameter type mismatch);
    ...
    }
    ...
  }
}
```



Assignment 2: Semantic Analysis

- Part 1: Symbol Table construction
  - AST + ST is input to the code generation

- Part 2: Semantic Analysis
  - Type analysis: expressions, statements, method declaration, method calls, array access, length member for arrays, …
  - Check for undeclared identifiers, check for duplicate identifiers, …