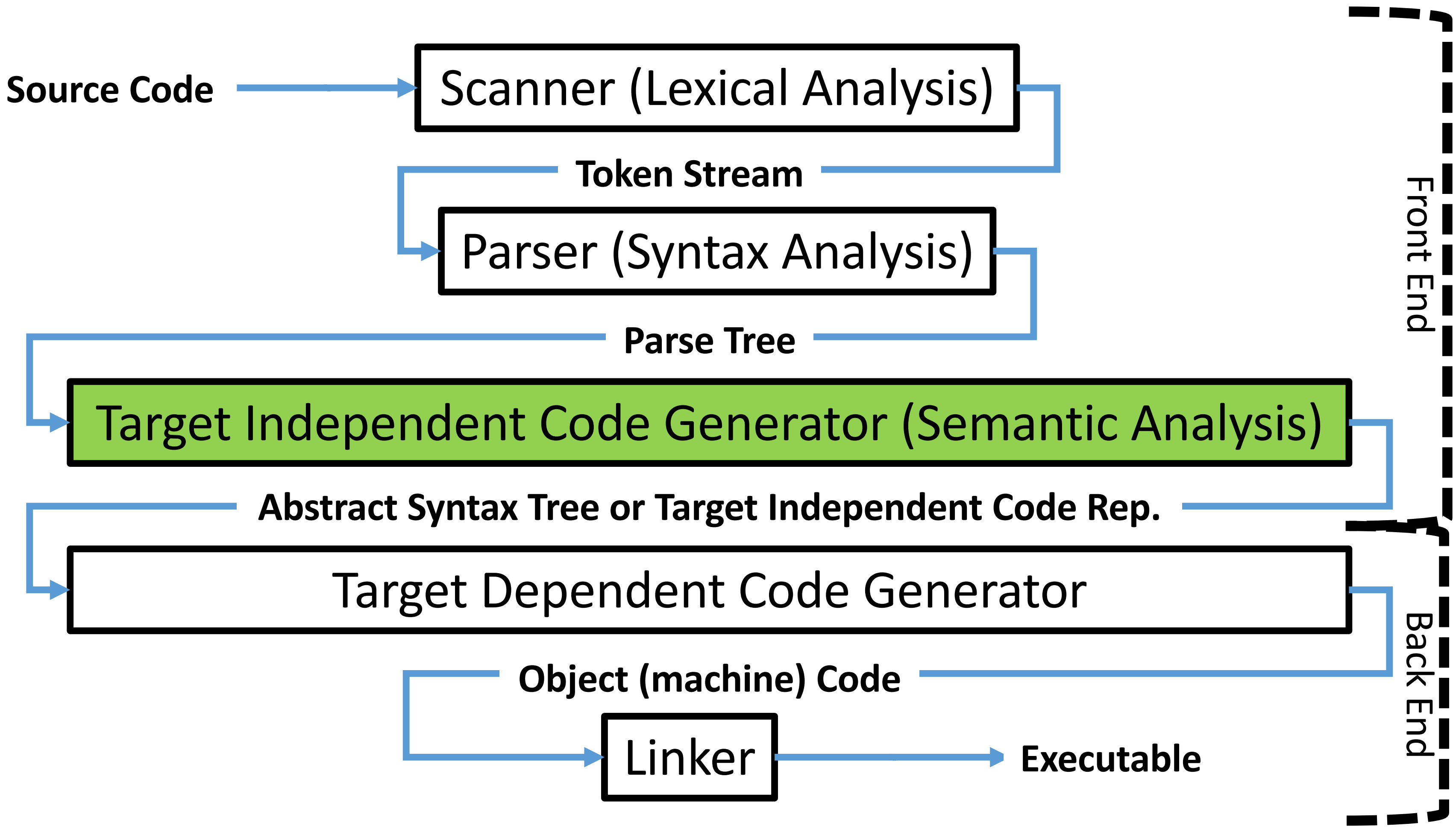
Symbol Tables and Code Generation

CSCI 3136: Principles of Programming Languages

Agenda

- Announcements
  - Assignment 5 due June 27
  - Midterm returned: mean 70%, median 74%
    - Well done!
- Readings: Read Chapter 4
- Lecture Contents
  - Tophat
  - S-Attributed and L-Attributed Grammars
  - Symbol Tables
  - Code Generation

Recall: Phases of Compilation



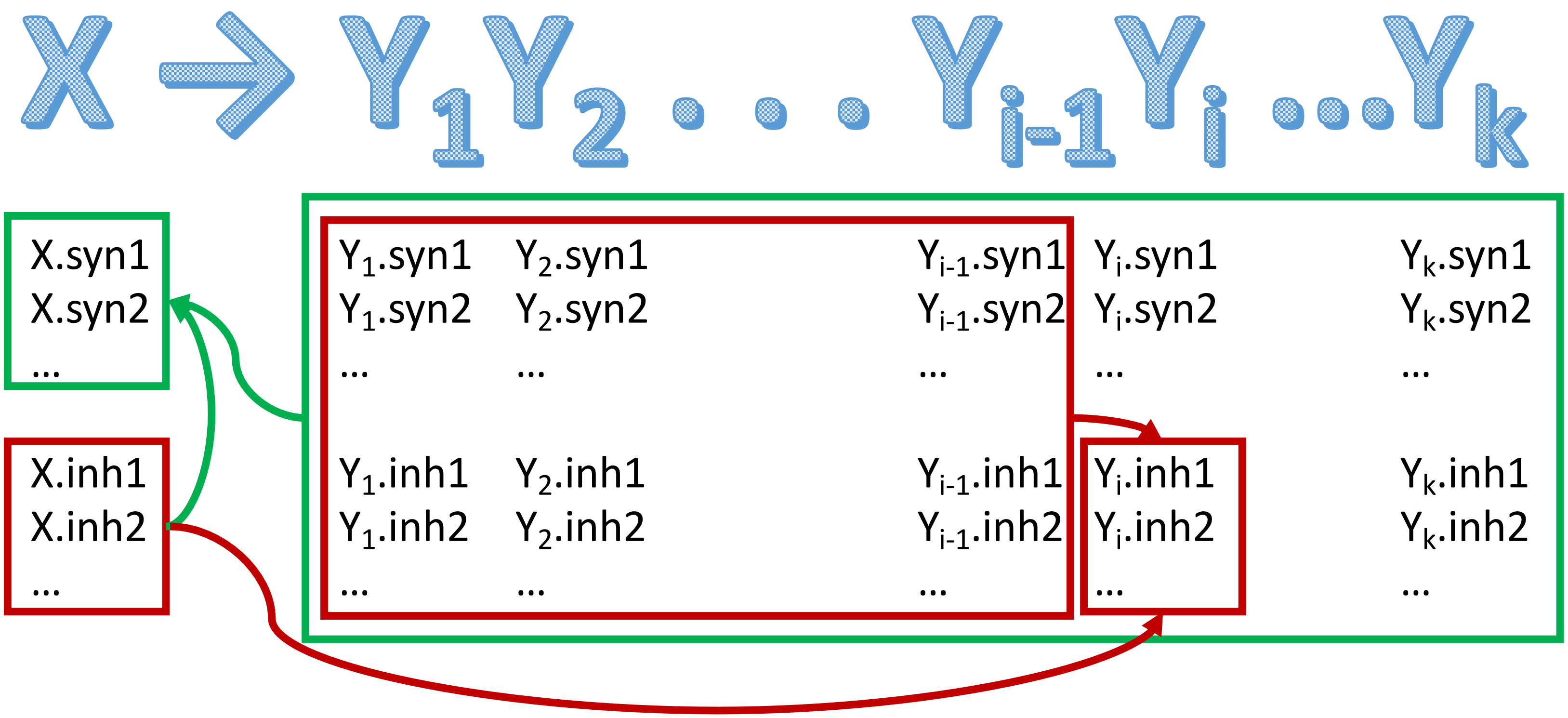
Recap

- Parse trees can be annotated or decorated with attributes and rules, which are executed as the tree is traversed.
- Synthesized attributes
  - Attributes of LHS of production are computed from attributes of RHS
  - Attributes flow bottom-up in the parse tree.
- Inherited attributes
  - Attributes in RHS are computed from attributes of LHS and symbols in RHS preceding them.
  - Attributes flow top-down in the parse tree.

S-Attributed and L-Attributed Grammars

- S-attributed grammar
  - All attributes are synthesized.
  - Attributes flow bottom-up.
- L-attributed grammar
  - Variables have both inherited and synthetic attributes
  - For each production $X \rightarrow Y_1Y_2 \ldots Y_k$,
    - X.syn depends on
      - X.inh
      - Y_1.inh, Y_1.syn, Y_2.inh, Y_2.syn, . . . Y_k.inh, Y_k.syn
  - For all $1 \leq i \leq k$, Y_i.inh depends on
    - X.inh
    - Y_1.inh, Y_1.syn, Y_2.inh, Y_2.syn, . . . , Y_{i-1}.inh, Y_{i-1}.syn
- S-attributed grammars are a special case of L-attributed grammars.

Data Flow in L-Attributed Grammars



Computing L-Attributed Grammars

```
execute_rules( Node t, Node [] left_sibs ):
  # Don't use t.synthetic and t.parent.synthetic
  t.compute_inherited( t.parent, left_sibs )
```

1

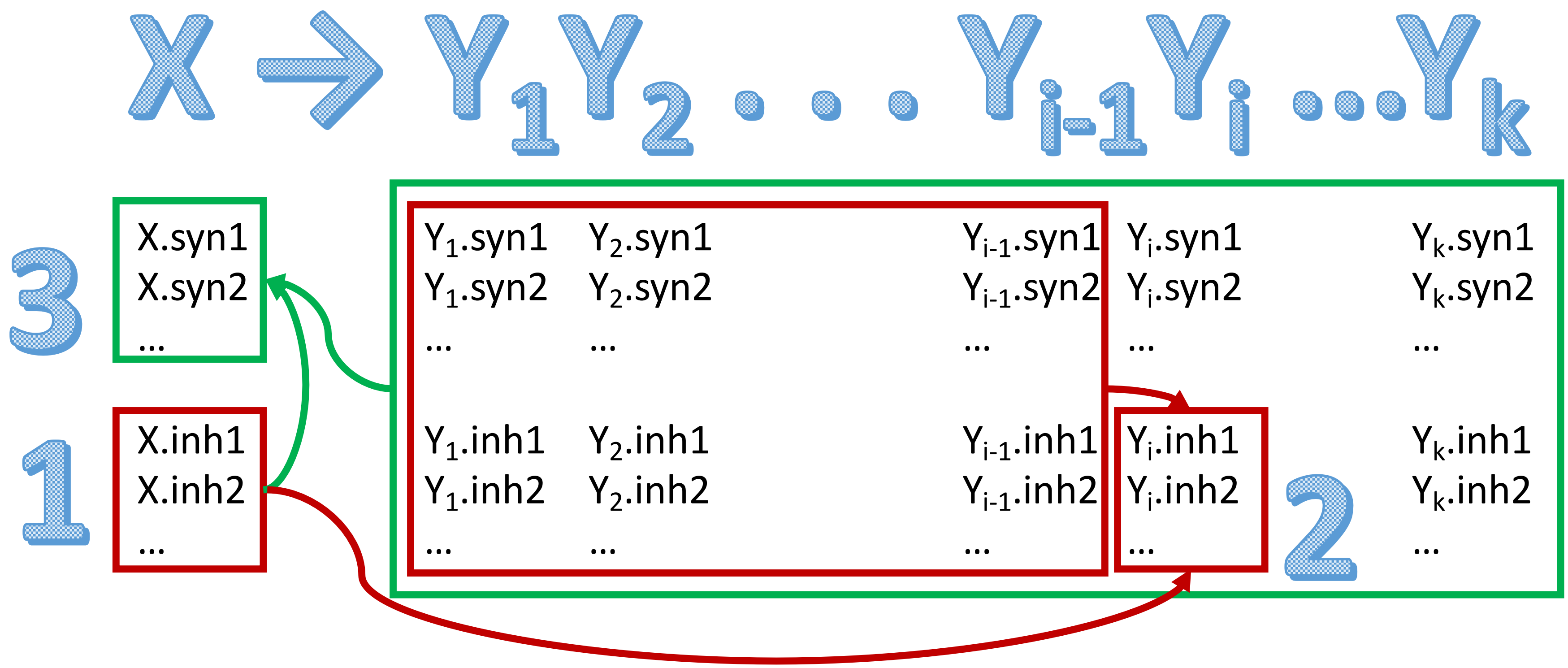
```
  children = []
  for each child of t:
    execute_rules( child, children )
    children.add( child )
```

2

```
  # Don't use t.synthetic and t.parent.synthetic
  t.compute_synthetic( children )
  return
```

3

Data Flow in L-Attributed Grammars



Motivation: Why are they useful?

- In many cases context free grammars that capture associativity rules are not LL(1)
- We can rewrite the grammars to be LL(1) but…
- Resulting grammars do no capture associativity rules
- So, use attribute (L-attributed) grammars to capture the associativity rules.

Example: Left Associative Grammar

- Grammar
  - E → E + T
  - E → E - T
  - E→T
  - T→ Int
- Parsing the expression

  5 − 2 + 3 illlustrates left associativity: (5 − 2) + 3
- This grammar is not LL(1)

Predictor Table	
Production	**Predictor Set**
E → E + T	{Int}
E → E - T	{Int}
E → T	{Int}
T→ Int	{Int}

Example: Refactored Grammar

- Grammar
  - E → T E’
  - E’ → ε
  - E’ → + T E’
  - E’ → - T E’
  - T→ Int
- Parsing the expression

  5 – 2 + 3 illlustrates wrong associativity: 5 – (2 + 3)
- This grammar is LL(1)

Predictor Table	
Production	**Predictor Set**
E → T E’	{Int}
E’ → ε	{ε}
E’ → + T E’	(+}
E’ → - T E’	(-}
T→ Int	{Int}

Use an L-Attributed Grammar to Fix Left Associativity

Idea: Carry forward the left most computed value to ensure left associativity.

- Try parsing: 5 - 2 + 3

Sym	Attributes
E	val : int
E'	val : int **tmp : int**
T	val : int
Int	val : String

Labeled CFG	Semantic Rules
E → T_1 E'_1	◂ **E_1'.tmp = T_1.val**; E.val = E'.val
E' → ε	◂ E'.val = E'.tmp
E' → + T_1 E'_1	◂ **E_1'.tmp = E'.tmp + T_1.val;** E'.val = E_1'.val
E' → - T_1 E'_1	◂ **E_1'.tmp = E'.tmp - T_1.val**; E'.val = E_1'.val
T→ Int_1	◂ T.val = Str2Int(Int_1.val)

Example: Error Checking

Labeled CFG	Semantic Rules
Assignment → $LValue_1$ ‘=‘ $Expr_1$	◃ *if not* assignable($Lvalue_1$.t, $Expr_1$.t), *error*
LValue → Id_1 $ArrIdx_1$	◃ *if not* declared(Id_1.name), *error* ◃ *if not* indexable(Id_1.name, $ArrIdx_1$.dim), *error*
ArrIdx → ε	◃ ArrIdx.dim = 0
ArrIdx → ‘[‘ $Expr_1$ ‘]’ $ArrIdx_1$	◃ *if not* isType($Expr_1$.t, Integer), *error* ◃ ArrIdx.dim = $ArrIdx_1$.dim + 1

Sym	Attributes
Assignment	
LValue	t : Type
Id	name : String
ArrIdx	dim : int
Expr	t : Type

Symbol Tables

- How do we keep track of a variable’s
  - Type (what is assignable to it)
  - Address (it’s location in memory)
  - Visibility (is it an automatic, static, or global variable)
  - Initialization (has the variable been initialized)
  - Scope
  - Etc.
- Idea: Store all symbols (identifiers)
  - Variables/Fields
  - Functions/Methods
  - Types/Classes/Structs
  - Etc

  in a symbol table.
- Definition: The symbol table is a set of tuples that include each symbol’s name and other properties.

  (name, symbol type, type, location, visibility, …)

Operations on a Symbol Table

- The primary operations are
  - `insert(name, properties)`
    - Inserts new symbol with given name and properties
  - `lookup(name)`
    - Finds the symbol with name
    - Returns a reference to the symbol's properties
- If the symbol is in the symbol table,
  - The symbol has been declared
  - The symbol's type is likely known
  - The symbol is visible in the current portion of code being analyzed
- Secondary operations include:
  - bool isDeclared(name)
  - bool isIndexable(name, dim)
- How do we implement a symbol table?

Symbol Table Implementation

- Use a hash table to map names (strings) to properties (tuples/structs/objects/etc)
- The insert() operation inserts (name, properties) into hash table.
- The lookup() operation gets name from hash table, which returns the properties of the symbol.

- How do we use the symbol table during semantic analysis?

Example: Variable Declarations

Labeled CFG	Semantic Rules
Function → $Type_1$ id_1 ($ParamList_1$) $Body_1$	◂ **…** ***$Body_1$.symtab = Function.symtab***
Body → { $Statements_1$ }	◂ … **$Statements_1$.symtab = Body.symtab**
Statement → $Statement_1$ $Statements_1$	◂ … **$Statement_1$.symtab = Statement.symtab** **$Statements_1$.symtab = $Statement_1$.symtab**
Statements → ε	
Statement → $VarDecl_1$	◂ … **$VarDecl_1$.symtab = Statement.symtab**
VarDecl → $Type_1$ id_1 $Initializer_1$	◂ … **$VarDecl_1$.symtab.insert(id_1.name, $Type_1$.typ)**
Initializer → = $Expr_1$	◂ …
Initializer → ε	◂ …

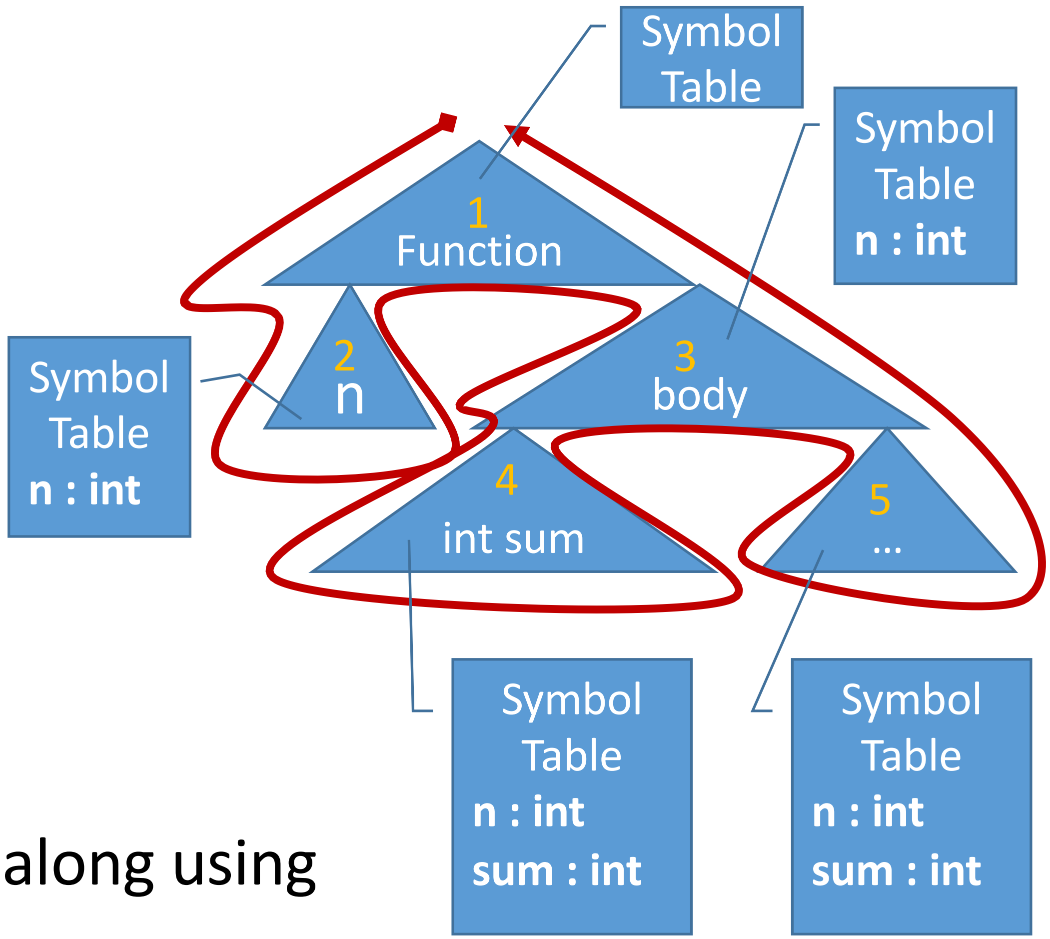
Sym	Attributes
Function	**symtab : SymTab**
Body	**symtab : SymTab**
Statements	**symtab : SymTab**
Statement	**symtab : SymTab**
VarDecl	**symtab : SymTab**

Consider a Function Definition

- Code

```
int fib(int n) {
   int sum
   …
}
```

- Order of evaluation:
  - Return type
  - Param List
  - Body
  - Variable Declaration
  - Rest of the body …
- Symbol table is passed along using inherited attributes



Questions to Address

- Most languages allow different functions/methods to use variables that have the same names.
  - How do we handle such conflicts?
  - Similar issue with classes
- Languages like Java allow methods to be called before they are declared.
  - How do we handle this?
- Parts of the symbol table are actually written as part of the compiled code. Why?

Code Generation

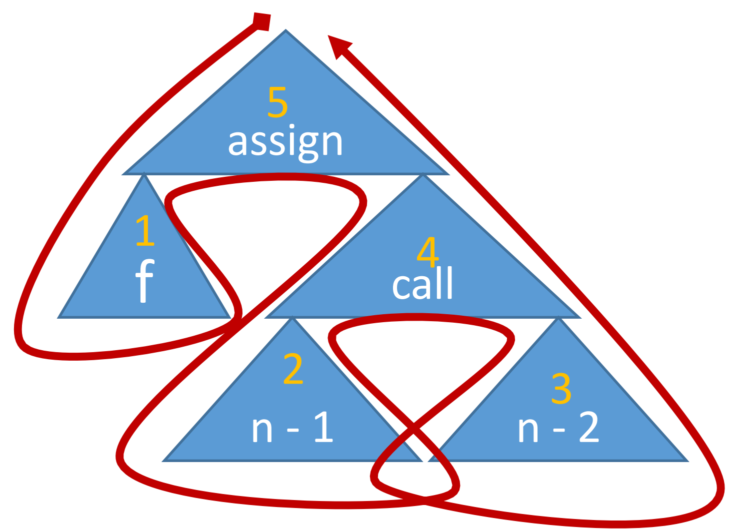
- Input Abstract Syntax Tree (AST)
- Task: Generate code for operations represented in the AST

  E.g. for Assignment Statements

    - Evaluate expression on RHS
    - Store in memory location corresponding to Lvalue
- Idea: Use semantic rules and attributes of the AST to generate code.
- But, how do we ensure that we generate code in the right order?

Consider a Function Call

- Code
  ```
  f = fib(n - 1, n - 2)
  ```
- Order of operations:
  - Determine location of **f**
  - Compute n – 1
  - Compute n – 2
  - Call fib()
  - Assign return value to **f**
- How does the abstract syntax tree look?
- How do we generate code in the right order?



Perform post-order traversal and use synthesized attributes

Code Generation Process

- Perform a post-order traversal of the AST
- Use synthesized attributes and semantic rules to generate platform independent code for the AST
- The post-order traversal ensures that the code will be properly ordered
- The code can then be optimized and platform specific code can be generated

Example: Generate Java Code for Expressions

Global Variable

	Labeled CFG	Semantic Rules
1	$E \rightarrow E_1 + T_1$	E.tmp = tmpSeqNum++ *output*(“int tmp%d = tmp%d + %s;”, E.tmp, E_1.tmp, T_1.var)
2	$E \rightarrow E_1 - T_1$	E.tmp = tmpSeqNum++ *output*(“int tmp%d = tmp%d - %s;”, E.tmp, E_1.tmp, T_1.var)
3	$E \rightarrow T_1$	E.tmp = tmpSeqNum++ *output(* “int tmp%d = %s;”, E.tmp, T_1.var)
4	$T \rightarrow Id_1$	T.var = id_1.name

Sym	Attributes
E	tmp : int
T	var : String
Id	name : String

Try generating Java code for the expression: a + b - c

Example: Generate Java Code for Expressions without a Global Vars

	Labeled CFG	Semantic Rules
1	$E \rightarrow E_1 + T_1$	**E_1.seq = E.seq** E.tmp = E_1.tmp *+ 1* *output*(“int tmp%d = tmp%d + %s;”, $E.tmp_{syn}$ E_1.tmp, T_1.var)
2	$E \rightarrow E_1 - T_1$	**E_1.seq = E.seq** E.tmp = E_1.tmp *+ 1* *output*(“int tmp%d = tmp%d - %s;”, E.tmp, E_1.tmp, T_1.var)
3	$E \rightarrow T_1$	E.tmp = **E.seq** + 1 *output(* “int tmp%d = %s;”, E.tmp, T_1.var)
4	$T \rightarrow Id_1$	T.var = id_1.name

Sym	Attributes
E	**seq : int** tmp : int
T	var : String
Id	name : String

Try generating Java code for the expression: a + b - c

Example: Generate Low-Level Code for Expressions

	Labeled CFG	Semantic Rules
1	$E \rightarrow E_1 + T_1$	◂ E.addr = new TempVariable() *generate*(MOVE E_1.addr, R1) *generate*(MOVE T_1.addr, R2) *generate*(ADD R1, R2) *generate*(MOVE R1, E.addr)
2	$E \rightarrow E_1 - T_1$	◂ E.addr = new TempVariable() *generate*(MOVE E_1.addr, R1) *generate*(MOVE T_1.addr, R2) *generate*(SUB R1, R2) *generate*(MOVE R1, E.addr)
3	$E \rightarrow T_1$	◂ E.addr = T_1.addr
4	$T \rightarrow Id_1$	◂ T.addr = getLocation(id_1.name)
5	$T \rightarrow Integer_1$	◂ T.addr = new TempVariable() *generate*(MOVE $Integer_1$.val, R1) *generate*(MOVE R1, T.addr)

Sym	Attributes
E	addr : pointer
T	addr : pointer
Id	name : String
Integer	val : int
A	op : Operator

Try generating assembly code for the expression:
a + 2 - b