Topic 4: Abstract Syntax Symbol Tables

COS 320

Compiling Techniques

Princeton University
Spring 2016

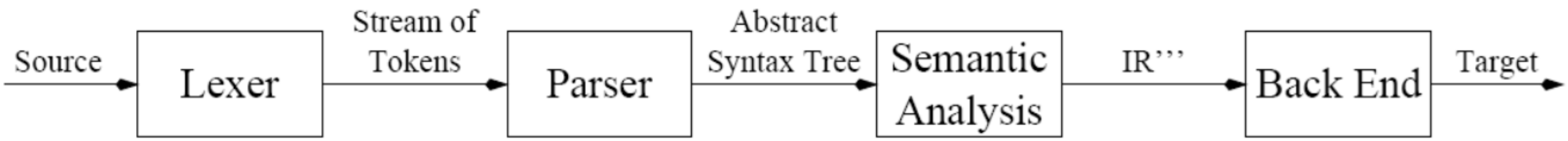
Lennart Beringer

Abstract Syntax

Can write entire compiler in ML-YACC specification.

- Semantic actions would perform type checking and translation to assembly.
- Disadvantages:
  1. File becomes too large, difficult to manage.
  2. Program must be processed in order in which it is parsed. Impossible to do global/inter-procedural optimization.

Alternative: Separate parsing from remaining compiler phases.

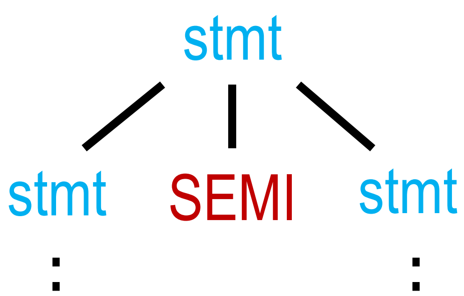


Parse Trees

We have been looking at **concrete** parse trees, in which

- inner nodes are nonterminals, leaf nodes are terminals
- children are labeled with the symbols in the RHS of the production

stmt → stmt SEMI stmt
stmt → …



Concrete parse trees are inconvenient to use, since they are cluttered with tokens containing no additional information:

- punctuation symbols (SEMI etc) needed to specify structure when writing code, but
- the tree structure already describes the program structure

Parse Tree Example

$P \rightarrow (\, S \,)$

$S \rightarrow S \, ; \, S$

$S \rightarrow \text{ID} := E$

$E \rightarrow \text{ID}$

$E \rightarrow \text{NUM}$

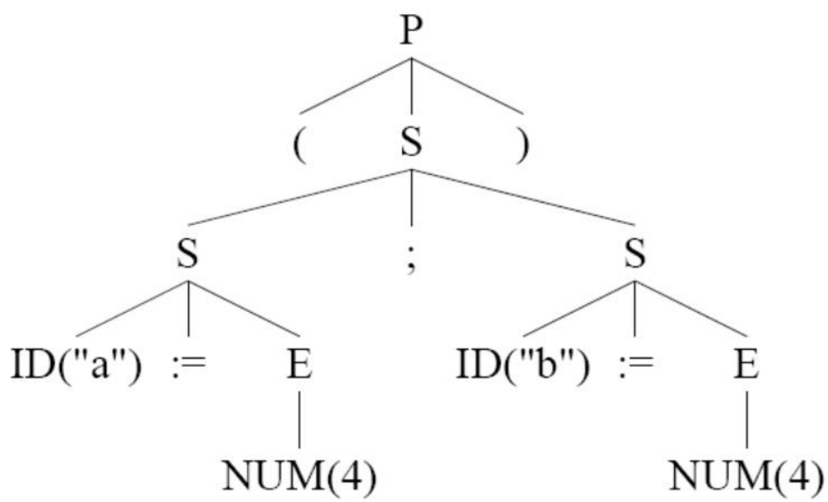
$E \rightarrow E + E$

$E \rightarrow E \,\text{-}\, E$

$E \rightarrow E * E$

$E \rightarrow E \, / \, E$

```
( a := 4 ; b := 5 )
```



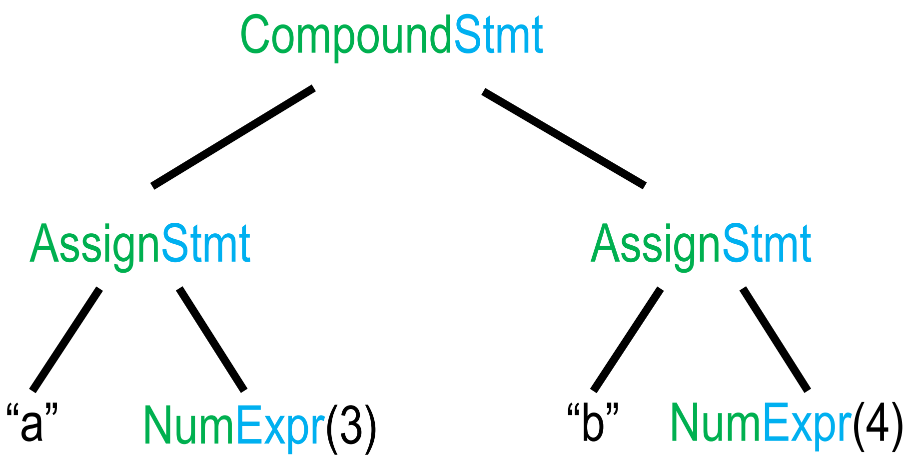
Type checker does not need “(” or “)” or “;”

Abstract parse trees (aka abstract syntac tree – AST)

- like concrete parse trees (e.g. inductive datatype, generated as semantic action by YACC)
- each syntactic category (expressions, statements,..) is represented as a separate datatype, with one constructor for each formation
- redundant punctuation symbols are left out

Abstract parse trees (aka abstract syntac tree – AST)

- like concrete parse trees (e.g. inductive datatype, generated as semantic action by YACC)
- each syntactic category (expressions, statements,..) is represented as a separate datatype, with one constructor for each formation
- redundant punctuation symbols are left out

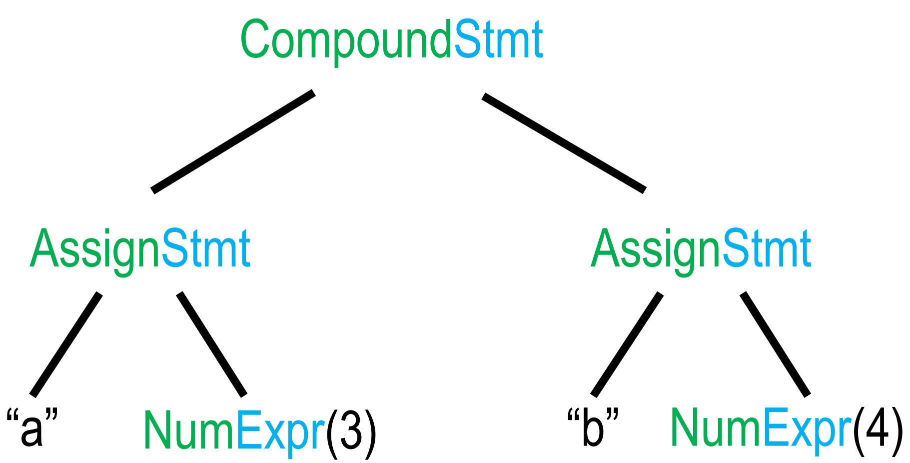



```
datatype stmt =
  CompoundStmt of stmt * stmt
| AssignStmt of string * expr;

datatype expr =
  NumExpr of int
| binopExpr of expr * binop * expr;
```

Abstract parse trees (aka abstract syntac tree – AST)

- like concrete parse trees (e.g. inductive datatype, generated as semantic action by YACC)
- each <u>syntactic category</u> (expressions, statements,..) is represented as a <u>separate datatype</u>, with one constructor for each formation
- redundant punctuation symbols are left out

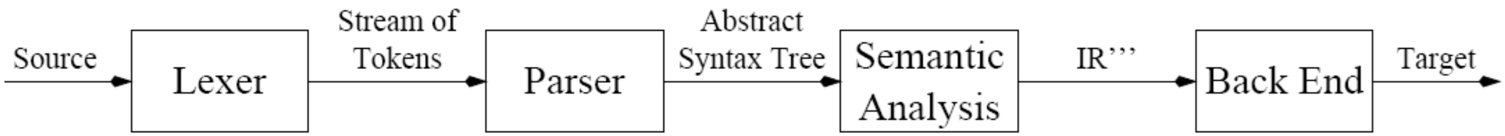



```
datatype stmt =
  CompoundStmt of stmt * stmt
| AssignStmt of string * expr;

datatype expr =
  NumExpr of int
| binopExpr of expr * binop * expr;
```

- <u>First approximation</u>: nonterminal ⇔ synt. category; CFG rule ⇔ constructor
- <u>But:</u> AST is internal interface between components of compiler, so AST design is up to compiler writer, not the language designer; may deviate from organization suggested by grammar/syntax

Semantic Analysis: Symbol Tables



- Semantic Analysis Phase:
  - Type check AST to make sure each expression has correct type
  - Translate AST into IR trees
- Main data structure used by semantic analysis: *symbol table*
  - Contains entries mapping identifiers to their bindings (e.g. type)
  - As new type, variable, function declarations encountered, symbol table augmented with entries mapping identifiers to bindings.
  - When identifier subsequently used, symbol table consulted to find info about identifier.
  - When identifier goes out of scope, entries are removed.

Symbol Table Example

```
function f (b:int, c:int)
= (print_int (b+c);
   let var j:= b
       var a := "x"
   in print (a);
      print_int (j)
   end;
   print_int (a)
 )
```

$\sigma_0 = \{a \mapsto int\}$

$\sigma_1 = \{b \mapsto int, c \mapsto int, a \mapsto int\}$

$\sigma_2 = \{j \mapsto int, b \mapsto int, c \mapsto int, a \mapsto int\}$

$\sigma_3 = \{a \mapsto string, j \mapsto int, b \mapsto int, c \mapsto int, a \mapsto int\}$

$\sigma_1 = \{b \mapsto int, c \mapsto int, a \mapsto int\}$

$\sigma_0 = \{a \mapsto int\}$

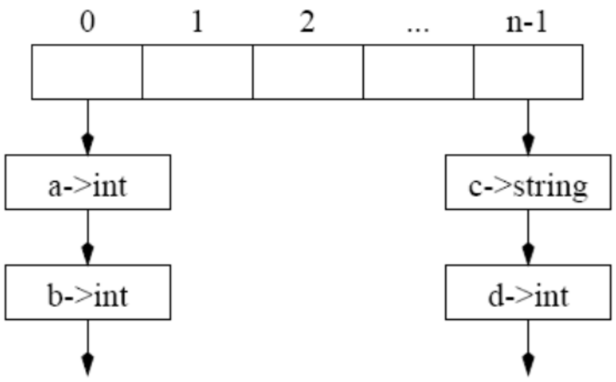
Symbol Table Implementation

- Imperative Style: (side effects)
  - Global symbol table
  - When beginning-of-scope entered, entries added to table using side-effects. (old table destroyed)
  - When end-of-scope reached, auxiliary info used to remove previous additions. (old table reconstructed)
- Functional Style: (no side effects)
  - When beginning-of-scope entered, *new* environment created by adding to old one, but old table remains intact.
  - When end-of-scope reached, retrieve old table.

Imperative Symbol Tables

Symbol tables must permit fast lookup of identifiers.

- *Hash Tables* - an array of *buckets*
- *Bucket* - linked list of entries (each entry maps identifier to binding)

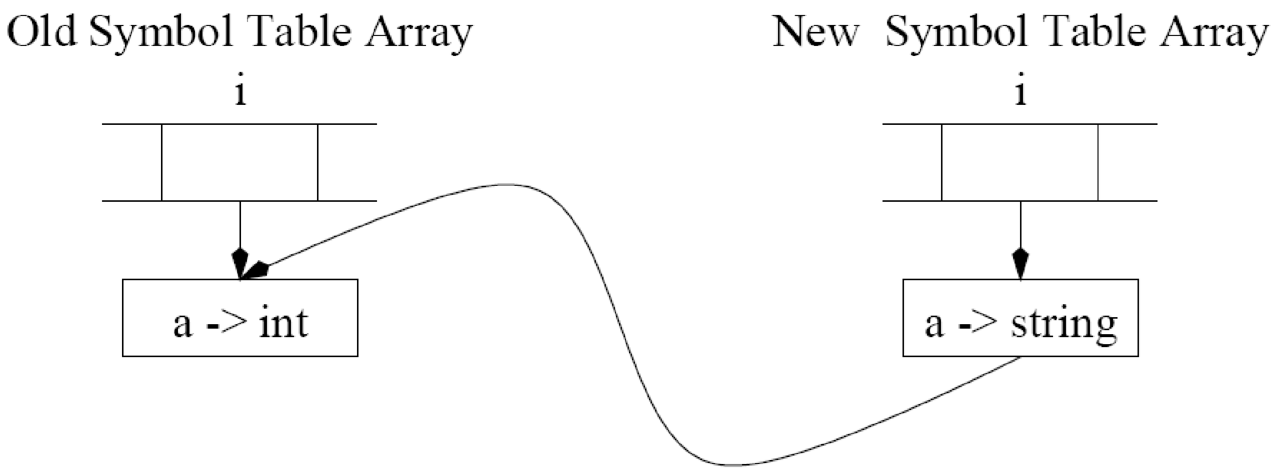


- Suppose we with to lookup entry for id i in symbol table:
  1. Apply *hash function* to key i to get array element $j \in [0, n-1]$.
  2. Traverse bucket in table[j] in order to find binding b.
     (table[x]: all entries whose keys hash to x)

Functional Symbol Tables

Hash tables not efficient for functional symbol tables.

Insert `a` $\mapsto$ `string` $\Rightarrow$ copy array, share buckets:



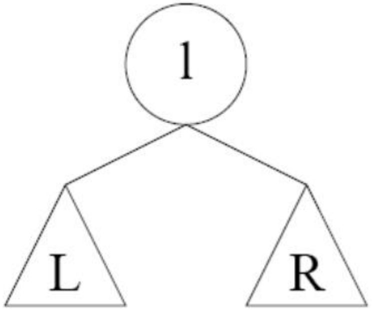
Not feasible to copy array each time entry added to table.

Association list (cf HW 1) not efficient (lookup and delete linear)

Functional Symbol Tables

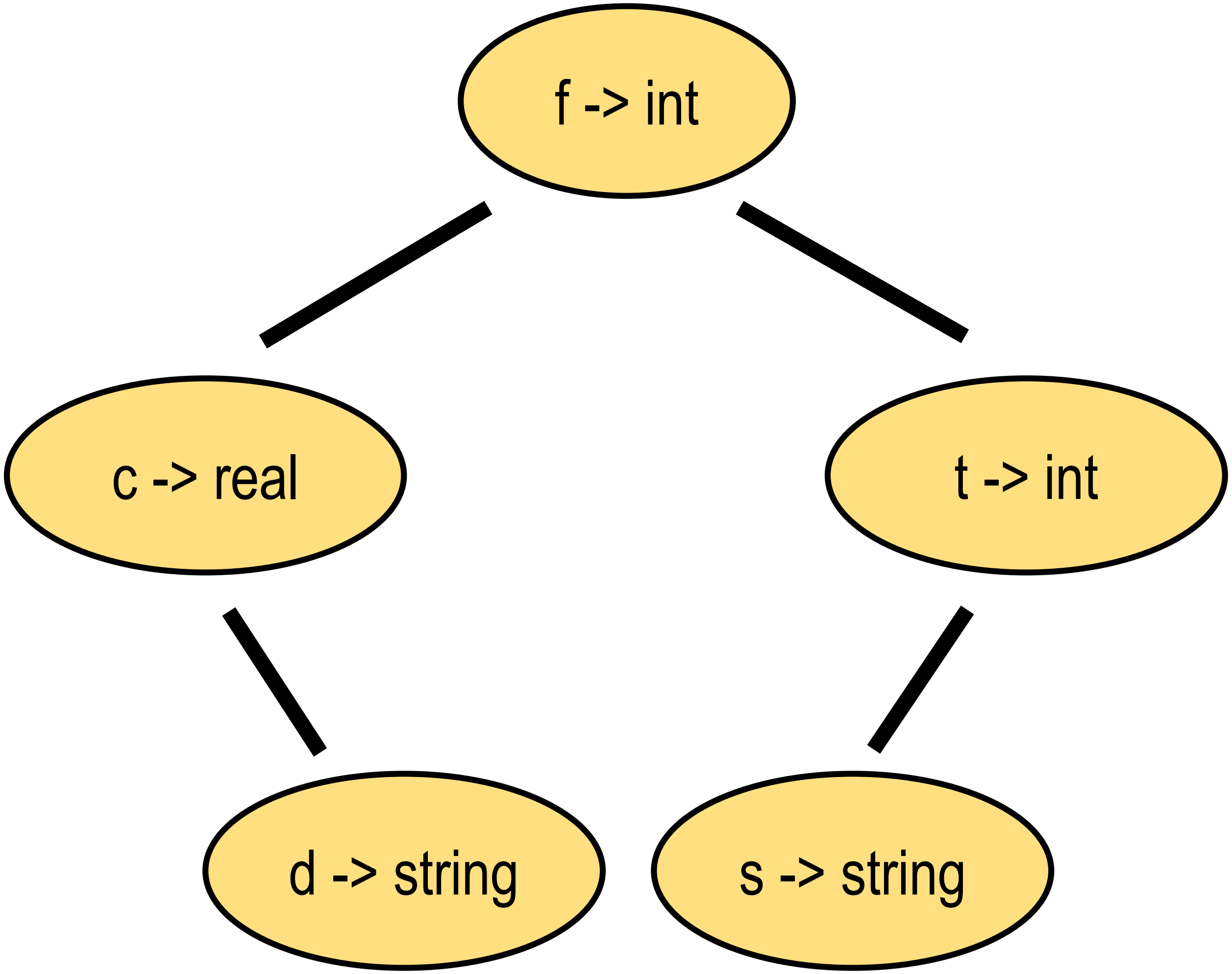
Better method: use *binary search trees (BSTs)*.

- Functional additions easy.
- Need "less than" ordering to build tree.
  - Each node contains mapping from identifier (key) to binding.
  - Use string comparison for "less than" ordering.
  - For all nodes $n \in L$, key(n) $<$ key(l)
    For all nodes $n \in R$, key(n) $>=$ key(l)

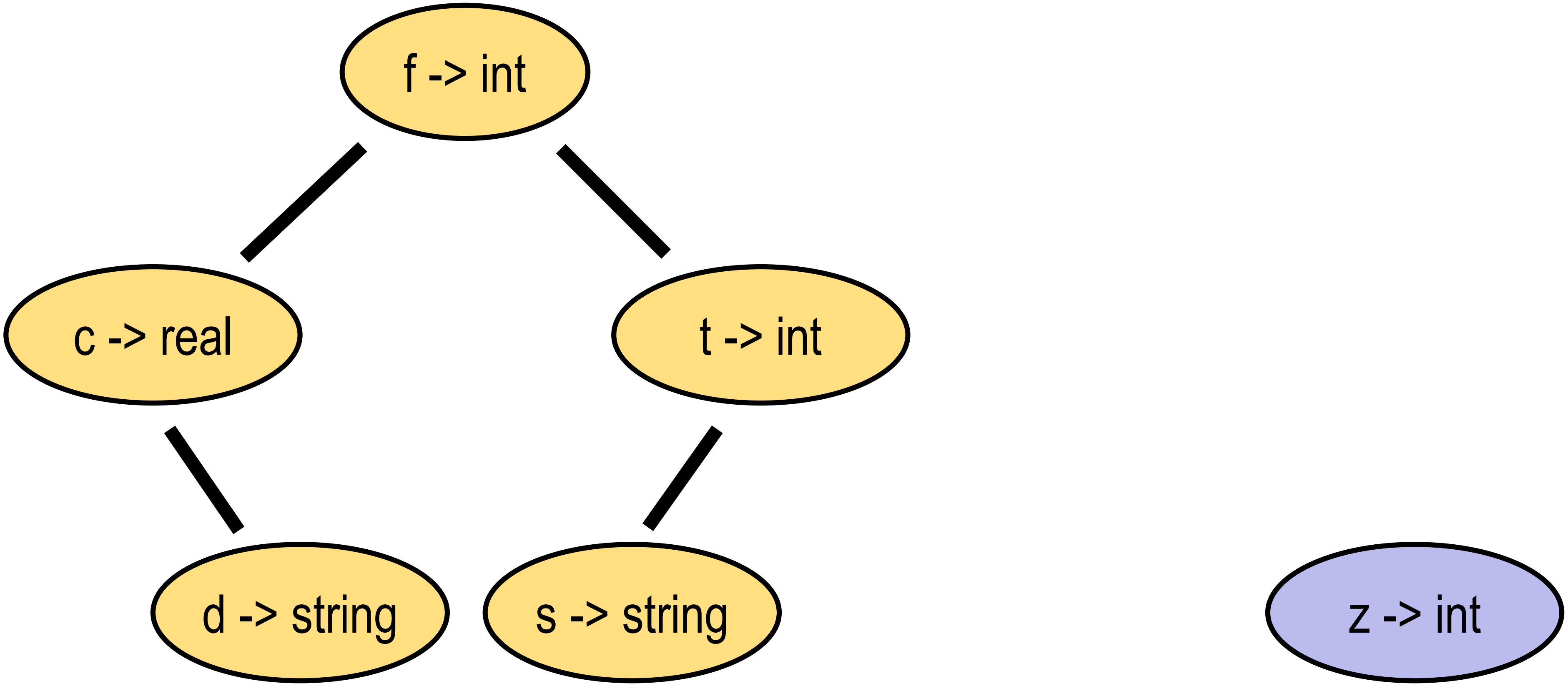


Functional Symbol Table using BST: lookup

Use the “less than” relation to navigate down the tree

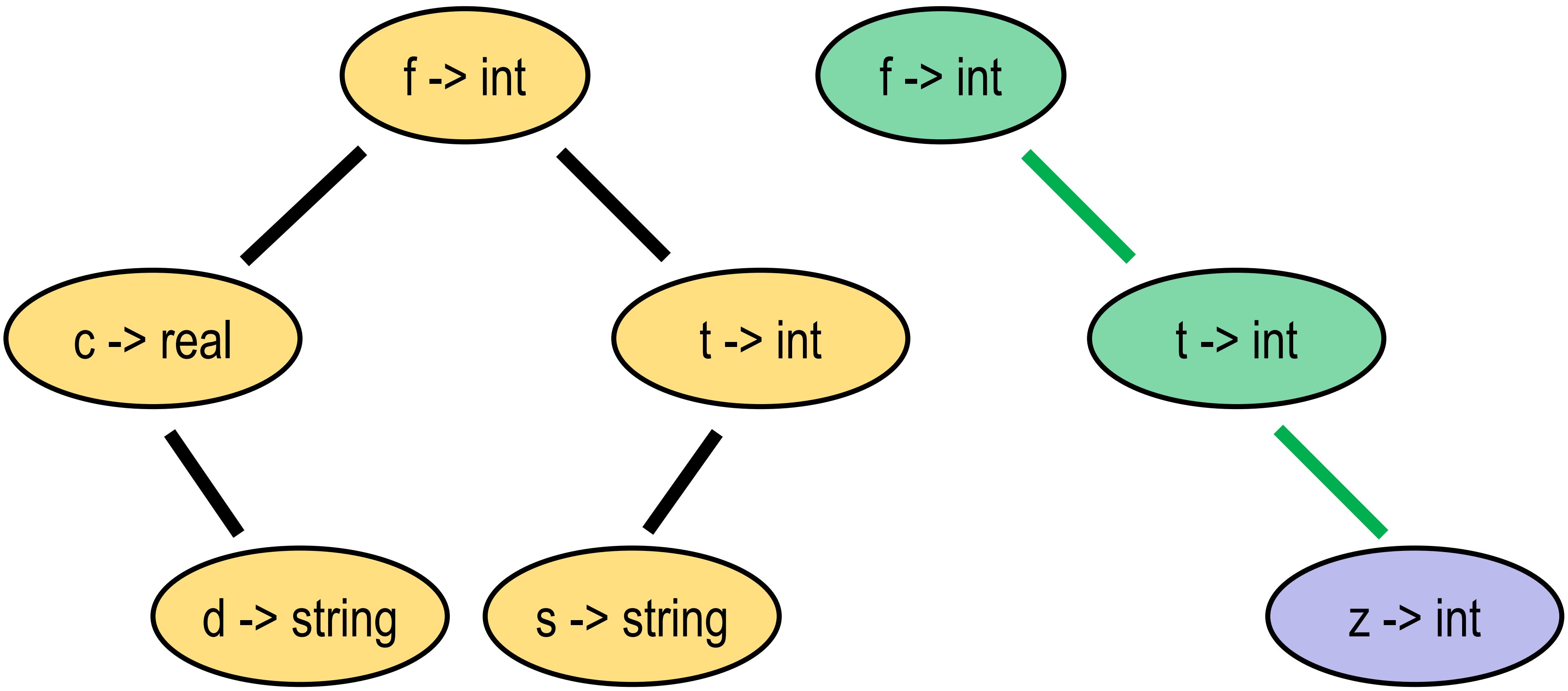


Functional Symbol Table using BST: insertion



Insertion of z-> int: 1. create node

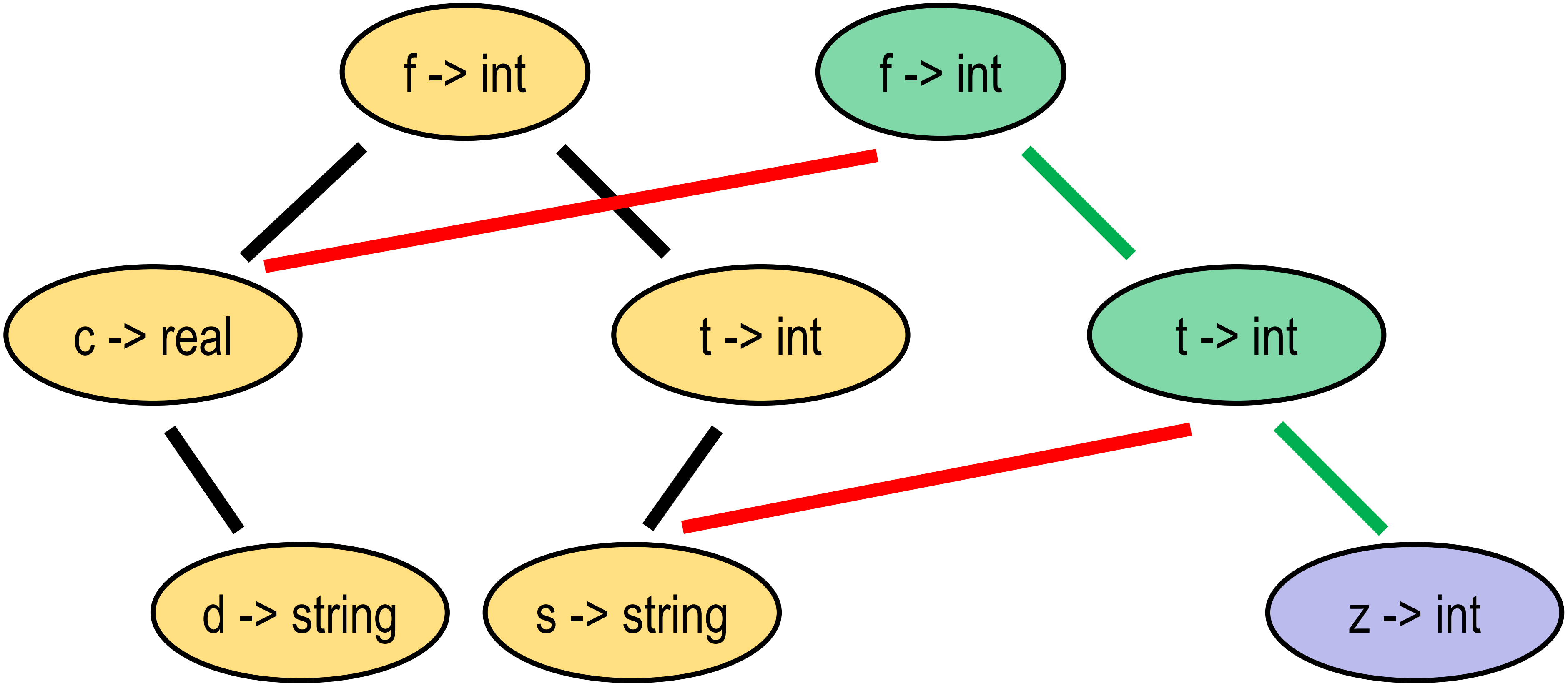
Functional Symbol Table using BST: insertion



Insertion of z-> int:

1. create node
2. “search” for z in old tree; copy ancestor nodes

Functional Symbol Table using BST: insertion



Insertion of z-> int:

1. create node
2. "search" for z in old tree; copy ancestor nodes
3. insert links to siblings in original (share subtree)