Parse Trees

- We have been looking at concrete parse trees.
  - Each internal node labeled with non-terminal.
  - Children labeled with symbols in RHS of production.
- Concrete parse trees inconvenient to use! Tree is cluttered with tokens containing no additional information.
  - Punctuation needed to specify structure when writing code, but
  - Tree structure itself cleanly describes program structure.

Parse Tree Example

\[
\begin{align*}
P & \rightarrow ( S ) \\
S & \rightarrow S ; S \\
S & \rightarrow ID := E \\
E & \rightarrow ID \\
E & \rightarrow NUM \\
E & \rightarrow E * E \\
E & \rightarrow E + E \\
E & \rightarrow E / E \\
\end{align*}
\]

\[
\begin{align*}
( & \quad a := 4 \; ; \; b := 5 )
\end{align*}
\]

\[
\text{Type checker does not need "(" or ")" or ","}
\]
Parse Tree Example

Solution: generate *abstract parse tree* (abstract syntax tree) - similar to concrete parse tree, except redundant punctuation tokens left out.

```
CompoundStmt
  AssignStmt
    ID("a") NUM(4)
  AssignStmt
    ID("b") NUM(4)
```

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(j)
    end
  print_int(a)
)
```

```
s_0 = {a \rightarrow \text{int}}

\text{let} \quad
s_1 = {b \rightarrow \text{int}, c \rightarrow \text{int}, a \rightarrow \text{int}}

\begin{align*}
\text{var } j &:= b \\
\text{var } a &:= "x"
\end{align*}

\begin{align*}
s_2 &= {j \rightarrow \text{int}, b \rightarrow \text{int}, c \rightarrow \text{int}, a \rightarrow \text{int}} \\
s_2 &= {a \rightarrow \text{string}, j \rightarrow \text{int}, b \rightarrow \text{int}, c \rightarrow \text{int}, a \rightarrow \text{int}}
\end{align*}

\begin{align*}
s_3 &= {b \rightarrow \text{int}, c \rightarrow \text{int}, a \rightarrow \text{int}} \\
s_4 &= {b \rightarrow \text{int}, c \rightarrow \text{int}, a \rightarrow \text{int}}
\end{align*}
```

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)

<table>
<thead>
<tr>
<th>( 0 )</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>...</th>
<th>( n-1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \sim \text{int} )</td>
<td>( c \sim \text{string} )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b \sim \text{int} )</td>
<td>( d \sim \text{int} )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Suppose we wish 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 \( \text{table}[j] \) in order to find binding \( b \).

**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 \( n \in L \), \( \text{key}(n) < \text{key}(l) \)
    - For all nodes \( n \in R \), \( \text{key}(n) \geq \text{key}(l) \)

**Functional Symbol Table Example**

Lookup:

- **Hash tables** not efficient for functional symbol tables.
  - Insert a \( \sim \text{string} \Rightarrow \text{copy array, share buckets:} \)

```
Old Symbol Table Array

New Symbol Table Array
```

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

insert $z \leftarrow \text{int}$, create node $z$, copy all ancestors of $z$: