Topic:

Semantic Analysis
Scanner and parser accept programs with correct syntax.

- not all programs that are syntactically correct make sense
- the compiler may be able to recognize some of these
  - these programs are rejected and reported as erroneous
  - the language definition defines what erroneous means
- semantic analyses are necessary that, for instance:
  - check that identifiers are known and where they are defined
  - check the type-correct use of variables
- semantic analyses are also useful to
  - find possibilities to “optimize” the program
  - warn about possibly incorrect programs

~ a semantic analysis annotates the syntax tree with attributes
Chapter 1: Attribute Grammars
Attribute Grammars

- many computations of the semantic analysis as well as the code generation operate on the syntax tree
- what is computed at a given node only depends on the type of that node (which is usually a non-terminal)
- we call this a local computation:
  - only accesses already computed information from neighbouring nodes
  - computes new information for the current node and other neighbouring nodes

**Definition** attribute grammar

An attribute grammar is a CFG extended by

- a set of attributes for each non-terminal and terminal
- local attribute equations

in order to be able to evaluate the attribute equations, all attributes mentioned in that equation have to be evaluated already

→ the nodes of the syntax tree need to be visited in a certain sequence
Example: Computation of the empty\(^r\) Attribute

Consider the syntax tree of the regular expression \((a|b)^*a(a|b)\):

\[\rightsquigarrow\] equations for empty\(^r\) are computed from bottom to top (aka bottom-up)
Implementation Strategy

- attach an attribute `empty` to every node of the syntax tree
- compute the attributes in a *depth-first post-order* traversal:
  - at a leaf, we can compute the value of `empty` without considering other nodes
  - the attribute of an inner node only depends on the attribute of its children
- the `empty` attribute is a *synthesized* attribute

in general:

**Definition**

An attribute at $N$ is called

- *inherited* if its value is defined in terms of attributes of $N$’s parent, siblings and/or $N$ itself (root $\leftrightarrow$ leaves)
- *synthesized* if its value is defined in terms of attributes of $N$’s children and/or $N$ itself (leaves $\rightarrow$ root)
Example: Attribute Equations for \textit{empty}

In order to compute an attribute \textit{locally}, specify attribute equations for each node depending on the \textit{type} of the node:

In the \textbf{Example} from earlier, we did that intuitively:

for leaves: \( r \equiv \begin{array}{c} i \end{array} x \) we define \( \text{empty}[r] = (x \equiv \epsilon) \).

otherwise:

\[
\begin{align*}
\text{empty}[r_1 \mid r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
\text{empty}[r_1^*] &= t \\
\text{empty}[r_1?] &= t
\end{align*}
\]
In general, for establishing attribute systems we need a flexible way to refer to parents and children:

We use consecutive indices to refer to neighbouring attributes:

- \( \text{attribute}_k[0] \): the attribute of the current root node
- \( \text{attribute}_k[i] \): the attribute of the \( i \)-th child \( (i > 0) \)

... the example, now in general formalization:

\[
\begin{align*}
x & : \quad \text{empty}[0] := (x \equiv \epsilon) \\
\mid & : \quad \text{empty}[0] := \text{empty}[1] \lor \text{empty}[2] \\
\cdot & : \quad \text{empty}[0] := \text{empty}[1] \land \text{empty}[2] \\
\ast & : \quad \text{empty}[0] := t \\
? & : \quad \text{empty}[0] := t
\end{align*}
\]
Observations

- the *local* attribute equations need to be evaluated using a *global* algorithm that knows about the dependencies of the equations
- in order to construct this algorithm, we need
  1. a sequence in which the nodes of the tree are visited
  2. a sequence within each node in which the equations are evaluated
- this *evaluation strategy* has to be compatible with the *dependencies* between attributes

We visualize the attribute dependencies $D(p)$ of a production $p$ in a *Local Dependency Graph*:

Let $p = N_0 \rightarrow N_1|N_2$ in

$$D(p) = \{ (\text{empty}[1], \text{empty}[0]), (\text{empty}[2], \text{empty}[0]) \}$$

$\rightsquigarrow$ arrows point in the direction of information flow
Observations

- In order to infer an evaluation strategy, it is not enough to consider the *local* attribute dependencies at each node.
- The evaluation strategy must also depend on the *global* dependencies, that is, on the information flow between nodes.

⚠️ The global dependencies change with each particular syntax tree.
- In the example, the parent node is always depending on children only.
  - A depth-first post-order traversal is possible.
- In general, variable dependencies can be much *more complex*. 
Simultaneous Computation of Multiple Attributes

Computing empty, first, next from regular expressions:

\[
\begin{align*}
S \rightarrow E : & \quad \text{empty}[0] := \text{empty}[1] \\
& \quad \text{first}[0] := \text{first}[1] \\
& \quad \text{next}[1] := \emptyset \\
\end{align*}
\]

\[
\begin{align*}
E \rightarrow x : & \quad \text{empty}[0] := (x \equiv \epsilon) \\
& \quad \text{first}[0] := \{x \mid x \neq \epsilon\}
\end{align*}
\]

\[
D(S \rightarrow E) : \\
\begin{array}{ccc}
f & \rightarrow & S \\
\uparrow & & \uparrow \\
f & \rightarrow & E \\
\uparrow & & \uparrow \\
f & \rightarrow & E \\
\end{array}
\]

\[
D(S \rightarrow E) = \{ (\text{empty}[1], \text{empty}[0]), (\text{first}[1], \text{first}[0]) \}
\]

\[
D(E \rightarrow x) : \\
\begin{array}{ccc}
f & \rightarrow & E \\
\uparrow & & \uparrow \\
\uparrow & & \uparrow \\
x & \rightarrow & n
\end{array}
\]

\[
D(E \rightarrow x) = \{ \}
\]
Regular Expressions: Rules for Alternative

\[ E \to E | E \] :

\[
\begin{align*}
\text{empty}[0] & := \text{empty}[1] \lor \text{empty}[2] \\
\text{first}[0] & := \text{first}[1] \lor \text{first}[2] \\
\text{next}[1] & := \text{next}[0] \\
\text{next}[2] & := \text{next}[0]
\end{align*}
\]

\[ D(E \to E | E) = \{
\begin{align*}
(\text{empty}[1], \text{empty}[0]),
(\text{empty}[2], \text{empty}[0]),
(\text{first}[1], \text{first}[0]),
(\text{first}[2], \text{first}[0]),
(\text{next}[0], \text{next}[2]),
(\text{next}[0], \text{next}[1])
\end{align*}
\]
Regular Expressions: Rules for Concatenation

\[
E \rightarrow E \cdot E
\]

\[
\begin{align*}
\text{empty}[0] & : = \text{empty}[1] \land \text{empty}[2] \\
\text{first}[0] & : = \text{first}[1] \cup (\text{empty}[1] \lor \text{first}[2] : \emptyset) \\
\text{next}[1] & : = \text{first}[2] \cup (\text{empty}[2] \lor \text{next}[0] : \emptyset) \\
\text{next}[2] & : = \text{next}[0]
\end{align*}
\]

\[
D(E \rightarrow E \cdot E) : \quad D(E \rightarrow E \cdot E) = \{(\text{empty}[1], \text{empty}[0]), (\text{empty}[2], \text{empty}[0]), (\text{empty}[2], \text{next}[1]), (\text{empty}[1], \text{first}[0]), (\text{first}[1], \text{first}[0]), (\text{first}[2], \text{first}[0]), (\text{first}[2], \text{next}[1]), (\text{next}[0], \text{next}[2]), (\text{next}[0], \text{next}[1])\}
\]
Regular Expressions: Rules for Kleene-Star and Option

\[ E \rightarrow E^* \]
- \( \text{empty}[0] := t \)
- \( \text{first}[0] := \text{first}[1] \)
- \( \text{next}[1] := \text{first}[1] \cup \text{next}[0] \)

\[ D(E \rightarrow E^*) : \]
- \( f \rightarrow e \rightarrow * \rightarrow n \)
- \( f \rightarrow e \rightarrow E \rightarrow n \)

\[ D(E \rightarrow E^*) = \{ (\text{first}[1], \text{first}[0]), (\text{first}[1], \text{next}[2]), (\text{next}[0], \text{next}[1]) \} \]

\[ E \rightarrow E? \]
- \( \text{empty}[0] := t \)
- \( \text{first}[0] := \text{first}[1] \)
- \( \text{next}[1] := \text{next}[0] \)

\[ D(E \rightarrow E?) : \]
- \( f \rightarrow e \rightarrow ? \rightarrow n \)
- \( f \rightarrow e \rightarrow E \rightarrow n \)

\[ D(E \rightarrow E?) = \{ (\text{first}[1], \text{first}[0]), (\text{next}[0], \text{next}[1]) \} \]
Challenges for General Attribute Systems

Static evaluation
Is there a static evaluation strategy, which is generally applicable?

- an evaluation strategy can only exist, if for any derivation tree the dependencies between attributes are acyclic
- it is $\text{DEXPTIME}$-complete to check for cyclic dependencies [Jazayeri, Odgen, Rounds, 1975]

Ideas
1. Let the User specify the strategy
2. Determine the strategy dynamically
3. Automate subclasses only
Subclass: Strongly Acyclic Attribute Dependencies

Idea: For all nonterminals $X$ compute a set $R(X)$ of relations between its attributes, as an overapproximation of the global dependencies between root attributes of every production for $X$.

Describe $R(X)$s as sets of relations, similar to $D(p)$ by

- setting up each production $X \rightarrow X_1 \ldots X_k$’s effect on the relations of $R(X)$
- compute effect on all so far accumulated evaluations of each rhs $X_i$’s $R(X_i)$
- iterate until stable
Subclass: Strongly Acyclic Attribute Dependencies

The 2-ary operator $L[i]$ re-decorates relations from $L$

$$L[i] = \{(a[i], b[i]) | (a, b) \in L\}$$

$\pi_0$ projects only onto relations between root elements only

$$\pi_0(S) = \{(a, b) | (a[0], b[0]) \in S\}$$

$[.]^\# \ldots$ root-projects the transitive closure of relations from the $L_i$s and $D$

$$[p]^\#(L_1, \ldots, L_k) = \pi_0((D(p) \cup L_1[1] \cup \ldots \cup L_k[k])^+)$$

$R$ maps symbols to relations (global attributes dependencies)

$$R(X) \supseteq (\bigcup\{[p]^\#(R(X_1), \ldots, R(X_k)) | p : X \rightarrow X_1 \ldots X_k\})^+ \ | \ p \in P$$

$$R(X) \supseteq \emptyset \ | \ X \in (N \cup T)$$

Strongly Acyclic Grammars

The system of inequalities $R(X)$
- characterizes the class of strongly acyclic Dependencies
- has a unique least solution $R^*(X)$ (as $[.]^\#$ is monotonic)
Subclass: Strongly Acyclic Attribute Dependencies

Strongly Acyclic Grammars

If all $D(p) \cup \mathcal{R}^*(X_1)[1] \cup \ldots \cup \mathcal{R}^*(X_k)[k]$ are acyclic for all $p \in G$, $G'$ is strongly acyclic.

Idea: we compute the least solution $\mathcal{R}^*(X)$ of $\mathcal{R}(X)$ by a fixpoint computation, starting from $\mathcal{R}(X) = \emptyset$. 
Example: Strong Acyclic Test

Given grammar $S \rightarrow L$, $L \rightarrow a \mid b$. Dependency graphs $D_p$:

```
S
  ^
 / \  \\
/   \  \\
L    \\
  ^
 /  \\
/   \\
 a  \\
```

```
S
  ^
 / \  \\
/   \  \\
L    \\
  ^
 /  \\
/   \\
b  \\
```
Example: Strong Acyclic Test

Start with computing $\mathcal{R}(L) = [L \rightarrow a]^{\#}() \sqcup [L \rightarrow b]^{\#}()$:

1. terminal symbols do not contribute dependencies
2. transitive closure of all relations in $(D(L \rightarrow a))^+$ and $(D(L \rightarrow b))^+$
3. apply $\pi_0$
4. $\mathcal{R}(L) = \{(k, j), (i, h)\}$
Example: Strong Acyclic Test

Continue with $\mathcal{R}(S) = \llbracket S \rightarrow L \rrbracket^\#(\mathcal{R}(L))$:

1. re-decorate and embed $\mathcal{R}(L)[1]$  
2. transitive closure of all relations $\left( D(S \rightarrow L) \cup \{(k[1], j[1])\} \cup \{(i[1], h[1])\}\right)^+$  
3. apply $\pi_0$  
4. $\mathcal{R}(S) = \{\}$  

check for cycles!
Strong Acyclic and Acyclic

The grammar $S \rightarrow L$, $L \rightarrow a \mid b$ has only two derivation trees which are both acyclic:

It is not strongly acyclic since the over-approximated global dependence graph for the non-terminal $L$ contributes to a cycle when computing $R(S)$:
Possible strategies:

1. let the *user* define the evaluation order
2. *automatic* strategy based on the dependencies
3. consider a *fixed* strategy and only allow an attribute system that can be evaluated using this strategy
Linear Order from Dependency Partial Order

Possible *automatic* strategies:

- **demand-driven evaluation**
  - start with the evaluation of any required attribute
  - if the equation for this attribute relies on as-of-yet unevaluated attributes, evaluate these recursively

- **evaluation in passes**
  for each pass, pre-compute a global strategy to visit the *nodes* together with a local strategy for evaluation *within each node* type
  ~ *minimize* the number of *visits* to each node
Example: Demand-Driven Evaluation

Compute $\text{next}$ at leaves $a_2$, $a_3$ and $b_4$ in the expression $(a \mid b)^* a (a \mid b)$:

$$| : \text{next}[1] := \text{next}[0]$$
$$\text{next}[2] := \text{next}[0]$$

$$\cdot : \text{next}[1] := \text{first}[2] \cup (\text{empty}[2] \ ? \text{next}[0] : \emptyset)$$
$$\text{next}[2] := \text{next}[0]$$

![Diagram of the expression with annotated nodes and values]
Demand-Driven Evaluation

Observations

- each node must contain a pointer to its parent
- *only required* attributes are evaluated
- the evaluation sequence depends – in general – on the actual syntax tree
- the algorithm must track which attributes it has already evaluated
- the algorithm may visit nodes more often than necessary

⇒ the algorithm is not local

In principle:
- evaluation strategy is dynamic: difficult to debug
- usually all attributes in all nodes are required

⇒ computation of all attributes is often cheaper

⇒ perform evaluation in *passes*
Implementing State

Problem: In many cases some sort of state is required.
Example: numbering the leaves of a syntax tree
Example: Implementing Numbering of Leafs

Idea:
- use helper attributes pre and post
- in pre we pass the value for the first leaf down (inherited attribute)
- in post we pass the value of the last leaf up (synthesized attribute)

root:
- pre[0] := 0
- pre[1] := pre[0]
- post[0] := post[1]

node:
- pre[1] := pre[0]
- post[0] := post[2]

leaf:
- post[0] := pre[0] + 1
the attribute system is apparently strongly acyclic

each node computes

- the inherited attributes before descending into a child node (corresponding to a pre-order traversal)
- the synthesized attributes after returning from a child node (corresponding to post-order traversal)

Definition L-Attributed Grammars

An attribute system is *L*-attributed, if for all productions $S \rightarrow S_1 \ldots S_n$ every inherited attribute of $S_j$ where $1 \leq j \leq n$ only depends on

1. the attributes of $S_1, S_2, \ldots S_{j-1}$ and
2. the inherited attributes of $S$. 
L-Attributation

Background:

- the attributes of an $L$-attributed grammar can be evaluated during parsing
- important if no syntax tree is required or if error messages should be emitted while parsing
- example: pocket calculator

$L$-attributed grammars have a fixed evaluation strategy:

a single depth-first traversal

- in general: partition all attributes into $\mathcal{A} = A_1 \cup \ldots \cup A_n$ such that for all attributes in $A_i$
  - the attribute system is $L$-attributed
- perform a depth-first traversal for each attribute set $A_i$

$\leadsto$ craft attribute system in a way that they can be partitioned into few $L$-attributed sets
Practical Applications

- Symbol tables, type checking/inference, and simple code generation can all be specified using $L$-attributed grammars.
- Most applications annotate syntax trees with additional information.
- The nodes in a syntax tree usually have different types that depend on the non-terminal that the node represents.
- The different types of non-terminals are characterized by the set of attributes with which they are decorated.

Example: Def-Use Analysis

- A statement may have two attributes containing valid identifiers: one ingoing (inherited) set and one outgoing (synthesised) set.
- An expression only has an ingoing set.
Implementation of Attribute Systems via a Visitor

- class with a method for every non-terminal in the grammar
  ```java
  public abstract class Regex {
    public abstract void accept(Visitor v);
  }
  ```

- attribute-evaluation works via pre-order / post-order callbacks
  ```java
  public interface Visitor {
    default void pre(OrEx re) {}
    default void pre(AndEx re) {}
    ...
    default void post(OrEx re) {}
    default void post(AndEx re){}
  }
  ```

- we pre-define a depth-first traversal of the syntax tree
  ```java
  public class OrEx extends Regex {
    Regex l,r;
    public void accept(Visitor v) {
      v.pre(this); l.accept(v); v.inter(this);
      r.accept(v); v.post(this);
    }
  }
  ```
Example: Leaf Numbering

```java
public abstract class AbstractVisitor implements Visitor {
    public void pre (OrEx re){ pr(re); }
    public void pre (AndEx re){ pr(re); }
    ... /* redirecting to default handler for bin exprs */
    public void post(OrEx re){ po(re); }
    public void post(AndEx re){ po(re); }
    abstract void po(BinEx re);
    abstract void in(BinEx re);
    abstract void pr(BinEx re);
}

public class LeafNum extends AbstractVisitor {
    public Map<Regex,Integer> pre = new HashMap<>();
    public Map<Regex,Integer> post = new HashMap<>();
    public LeafNum (Regex r) { pre .put(r,0); r.accept(this); }
    public void pre(Const r) { post.put(r, pre .get(r)+1); }
    public void pr (BinEx r) { pre .put(r.l, pre .get(r)); }
    public void in (BinEx r) { pre .put(r.r, post.get(r.l)); }
    public void po (BinEx r) { post.put(r, post.get(r.r)); }
}
```
Chapter 2:
Decl-Use Analysis
Symbol Bindings and Visibility

Consider the following Java code:

```java
void foo() {
    int a;
    while (true) {
        double a;
        a = 0.5;
        write(a);
        break;
    }
    a = 2;
    bar();
    write(a);
}
```

- each *declaration* of a variable `v` causes memory allocation for `v`
- using `v` requires knowledge about its memory location
  → determine the declaration `v` is *bound* to

- a binding is not *visible* when a local declaration of the same name is in scope

  *in the example* the definition of `A` is shadowed by the *local definition* in the loop body
Scope of Identifiers

```c
void foo() {
    int A;
    while (true) {
        double A;
        A = 0.5;
        write(A);
        break;
    }
    A = 2;
    bar();
    write(A);
}
```

administration of identifiers can be quite complicated...
Resolving Identifiers

**Observation:** each identifier in the AST must be translated into a memory access

**Problem:** for each identifier, find out what memory needs to be accessed by providing *rapid* access to its *declaration*

**Idea:**

1. *rapid* access: replace every identifier by a *unique* integer
   - integers as keys: comparisons of integers is faster

2. link each usage of a variable to the *declaration* of that variable
   - for languages without explicit declarations, create declarations when a variable is first encountered
Rapid Access: Replace Strings with Integers

Idea for Algorithm:

Input: a sequence of strings
Output: sequence of numbers

1. table that allows to retrieve the string that corresponds to a number
2. Apply this algorithm on each identifier during *scanning*.

Implementation approach:

1. count the number of new-found identifiers in `int count`
2. maintain a *hashtable* $S : \text{String} \rightarrow \text{int}$ to remember numbers for known identifiers

We thus define the function:

```cpp
int indexForIdentifier(String w) {
    if (S(w) ≡ undefined) {
        S = S ⊕ \{w ↦ count\};
        return count++;
    } else return S(w);
}
```
Implementation: Hashtables for Strings

1. allocate an array $M$ of sufficient size $m$
2. choose a hash function $H : \text{String} \rightarrow [0, m - 1]$ with:
   - $H(w)$ is cheap to compute
   - $H$ distributes the occurring words equally over $[0, m - 1]$

Possible generic choices for sequence types $(\vec{x} = \langle x_0, \ldots x_{r-1} \rangle)$:

$$
H_0(\vec{x}) = (x_0 + x_{r-1}) \mod m \\
H_1(\vec{x}) = (\sum_{i=0}^{r-1} x_i \cdot p^i) \mod m \\
H_1(\vec{x}) = (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{r-1} \ldots)) \mod m
$$

for some prime number $p$ (e.g. 31)

✗ The hash value of $w$ may not be unique!
   - Append $(w, i)$ to a linked list located at $M[H(w)]$
   - Finding the index for $w$, we compare $w$ with all $x$ for which $H(w) = H(x)$

✓ access on average:
   - insert: $O(1)$
   - lookup: $O(1)$
Example: Replacing Strings with Integers

Input:

<table>
<thead>
<tr>
<th>Peter</th>
<th>Piper</th>
<th>picked</th>
<th>a</th>
<th>peck</th>
<th>of</th>
<th>pickled</th>
<th>peppers</th>
</tr>
</thead>
</table>

If | Peter | Piper | picked | a | peck | of | pickled | peppers |

wheres | the | peck | of | pickled | peppers | Peter | Piper | picked |

Output:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

and

<table>
<thead>
<tr>
<th>0</th>
<th>Peter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Piper</td>
</tr>
<tr>
<td>2</td>
<td>picked</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>peck</td>
</tr>
<tr>
<td>5</td>
<td>of</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>pickled</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>peppers</td>
</tr>
<tr>
<td>8</td>
<td>If</td>
</tr>
<tr>
<td>9</td>
<td>wheres</td>
</tr>
<tr>
<td>10</td>
<td>the</td>
</tr>
</tbody>
</table>

Hashtable with $m = 7$ and $H_0$:

If | 8 | the | 10 |

pickled | 6 | peck | 4 | picked | 2 |

of | 5 | wheres | 9 | peppers | 7 |

Piper | 1 | Peter | 0 | a | 3 |
Refer Uses to Declarations: Symbol Tables

Check for the correct usage of variables:

- Traverse the syntax tree in a suitable sequence, such that
  - each declaration is visited before its use
  - the currently visible declaration is the last one visited

  ⇔ perfect for an L-attributed grammar
  - equation system for basic block must add and remove identifiers

- for each identifier, we manage a stack of declarations
  1. if we visit a declaration, we push it onto the stack of its identifier
  2. upon leaving the scope, we remove it from the stack

- if we visit a usage of an identifier, we pick the top-most declaration from its stack
- if the stack of the identifier is empty, we have found an undeclared identifier
Example: Decl-Use Analysis via Table of Stacks

```c
void f()
{
    int a, b;
    b = 5;
    if (b>3) {
        int a, c;
        a = 3;
        c = a + 1;
        b = c;
    } else {
        int c;
        c = a + 1;
        b = c;
    }
    b = a + b;
}
```

d declaration
b basic block
a assignment
Alternative Implementations for Symbol Tables

- When using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient.

\[
\begin{array}{c}
| a \\
| c \\
| a \\
| b \\
\end{array} \quad \begin{array}{c}
| c \\
| a \\
| b \\
\end{array}
\]

In front of if-statement \hspace{1cm} then-branch \hspace{1cm} else-branch

- Instead of lists of symbols, it is possible to use a list of hash tables; more efficient in large, shallow programs.

- An even more elegant solution: persistent trees (updates return fresh trees with references to the old tree where possible).

\sim a persistent tree \( t \) can be passed down into a basic block where new elements may be added, yielding a \( t' \); after examining the basic block, the analysis proceeds with the unchanged old \( t \).
Chapter 3:
Type Checking
Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type. For example: `int`, `void*`, `struct { int x; int y; }`.

Types are useful to

- manage memory
- to avoid certain run-time errors

In imperative and object-oriented programming languages a declaration has to specify a type. The compiler then checks for a type correct use of the declared entity.
Types are given using type-\textit{expressions}. The set of type expressions $T$ contains:

1. **base types**: `int`, `char`, `float`, `void`, ...
2. **type constructors** that can be applied to other types

example for type constructors in C:

- **structures**: `struct` \{ $t_1$ $a_1$; $\ldots$ $t_k$ $a_k$; \}
- **pointers**: $t$ *
- **arrays**: $t$ [ ]
  - the size of an array can be specified
  - the variable to be declared is written between $t$ and [$n$]
- **functions**: $t$ ($t_1$, $\ldots$, $t_k$)
  - the variable to be declared is written between $t$ and ($t_1$, $\ldots$, $t_k$)
  - in ML function types are written as: $t_1$ $\ast$ $\ldots$ $\ast$ $t_k$ $\rightarrow$ $t$
Type Checking

Problem:

Given: A set of type declarations $\Gamma = \{t_1 \; x_1; \ldots; t_m \; x_m;\}$

Check: Can an expression $e$ be given the type $t$?

Example:

```c
struct list { int info; struct list* next; };
int f(struct list* l) { return l; };
struct { struct list* c; }* b;
int* a[11];
```

Consider the expression:

```c
*a[f(b->c)]+2;
```
Type Checking using the Syntax Tree

Check the expression \( \ast a [ f (b \rightarrow c) ] + 2 \):

![Syntax Tree Diagram]

Idea:
- traverse the syntax tree bottom-up
- for each identifier, we lookup its type in \( \Gamma \)
- constants such as 2 or 0.5 have a fixed type
- the types of the inner nodes of the tree are deduced using typing rules
Type Systems

Formally: consider *judgements* of the form:

\[ \Gamma \vdash e : t \]

// (in the type environment \( \Gamma \) the expression \( e \) has type \( t \))

Axioms:

- **Const:** \( \Gamma \vdash c : t_c \)  
  \( (t_c \text{ type of constant } c) \)
- **Var:** \( \Gamma \vdash x : \Gamma(x) \)  
  \( (x \text{ Variable}) \)

Rules:

- **Ref:**  
  \[ \dfrac{\Gamma \vdash e : t}{\Gamma \vdash \& e : t^*} \]

- **Deref:**  
  \[ \dfrac{\Gamma \vdash e : t^*}{\Gamma \vdash * e : t} \]
Type Systems for C-like Languages

More rules for typing an expression: with subtyping relation $\leq$

Array:

\[
\frac{\Gamma \vdash e_1 : t * \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1[e_2] : t}
\]

Array:

\[
\frac{\Gamma \vdash e_1 : t[\,] \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1[e_2] : t}
\]

Struct:

\[
\frac{\Gamma \vdash e : \text{struct} \{ t_1 a_1; \ldots t_m a_m; \} \quad \Gamma \vdash e.a_i : t_i}{\Gamma \vdash e : t}
\]

App:

\[
\frac{\Gamma \vdash e : t(t_1, \ldots, t_m) \quad \Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash e(e_1, \ldots, e_m) : t}
\]

Op $\Box$:

\[
\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \Box e_2 : t_1 \sqcup t_2}
\]

Op $=$:

\[
\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 = e_2 : t_1 \quad t_2 \text{ can be converted to } \leq t_1}
\]

Explicit Cast:

\[
\frac{\Gamma \vdash e : t_2 \quad t_2 \text{ can be converted to } \leq t_1}{\Gamma \vdash (t_1) e : t_1}
\]
Example: Type Checking

Given expression \( \ast a[\ast f(b->c)] + 2 \) and 
\[
\Gamma = \{ \\
\text{struct list} \{ \text{int info; struct list* next; } \}; \\
\text{int } f(\text{struct list* } l); \\
\text{struct } \{ \text{struct list* } c; \} \ast b; \\
\text{int* a[11];} \\
\}
\]
Example: Type Checking – More formally:

\[ \Gamma = \{ \]

```c
struct list { int info; struct list* next; };
int f(struct list* l);
struct { struct list* c;}* b;
int* a[11];
```

\[ \}

\[ \text{VAR} \quad \Gamma \vdash b : \text{struct}\{\text{struct list *c;}}\}*
\]
\[ \text{DEREF} \quad \Gamma \vdash *b : \text{struct}\{\text{struct list *c;}}\)
\[ \Gamma \vdash (*b).c : \text{struct list*} \]
\[ \text{STRUCT} \]
\[ \text{VAR} \quad \Gamma \vdash f : _{\text{(t(struct list))}} \checkmark \quad \Gamma \vdash (*b).c : t\text{struct list*} \]
\[ \text{APP} \quad \Gamma \vdash f(b \rightarrow c) : \text{int} \checkmark \]
\[ \Gamma \vdash a[f(b \rightarrow c)] : \text{int*} \]
\[ \text{DEREF} \]
\[ \Gamma \vdash *a[f(b \rightarrow c)] : t\text{int} \]
\[ \Gamma \vdash *a[f(b \rightarrow c)] + 2 : t\text{int} \]
\[ \text{CONSTR} \]
\[ \Gamma \vdash 2 : t\text{int} \checkmark \]

but what do we do with \( \leq \)?
Equality of Types

Summary of Type Checking

- Choosing which rule to apply at an AST node is determined by the type of the child nodes
- determining the rule requires a check for \(\sim\) equality of types

**type equality** in C:

- `struct A {}` and `struct B {}` are considered to be different
  - \(\sim\) the compiler could re-order the fields of A and B independently *(not allowed in C)*
  - to extend an record A with more fields, it has to be embedded into another record:

```c
struct B {
    struct A;
    int field_of_B;
} extension_of_A;
```

- after issuing `typedef int C;` the types C and int are the same
Structural Type Equality

Alternative interpretation of type equality (*does not hold in C*):

*semantically*, two types $t_1, t_2$ can be considered as *equal* if they accept the same set of access paths.

Example:

```c
struct list {  
    int info;
    struct list* next;
}

struct list1 {  
    int info;
    struct {
        int info;
        struct list1* next;
    }* next;
}
```

Consider declarations `struct list* l` and `struct list1* l`. Both allow

```c
l->info  l->next->info
```

but the two declarations of `l` have unequal types in C.
Algorithm for Testing Structural Equality

Idea:

- track a set of equivalence queries of type expressions
- if two types are *syntactically* equal, we stop and report success
- otherwise, reduce the equivalence query to a several equivalence queries on (hopefully) *simpler* type expressions

Suppose that recursive types were introduced using type definitions:

```cpp
typedef A t
```

(we omit the $\Gamma$). Then define the following rules:
Rules for Well-Typedness

```c
struct {int info; A* next; } B

struct {int info; A* next; } struct {int info; ... *next; }

int int

A * ...

A struct {int info; B* next; }

struct {int info; A* next; } struct {int info; B* next; }

int int

A * B *

A B

struct {int info; A* next; } B
```
Example:

```c
typedef struct {
    int info;
    A * next;
} A

typedef struct {
    int info;
    struct {
        int info;
        B * next;
    } * next;
} B
```

We ask, for instance, if the following equality holds:

```c
struct {int info; A * next;} = B
```

We construct the following deduction tree:
Proof for the Example:

```c
typedef struct { int info; A * next; } A
typedef struct { int info; struct { int info; B * next; } * next; } B
```

```
struct { int info; A * next; } B
```

```
int int
```

```
A * ... *
```

```
A struct { int info; B * next; }
```

```
struct { int info; A * next; } struct { int info; B * next; }
```

```
int int
```

```
A * B *
```

```
A B
```

```
struct { int info; A * next; } B
```
Implementation

We implement a function that implements the equivalence query for two types by applying the deduction rules:

- if no deduction rule applies, then the two types are \textit{not equal}
- if the deduction rule for expanding a type definition applies, the function is called recursively with a \textit{potentially larger} type
- in case an equivalence query appears a second time, the types are \textit{equal by definition}

Termination

- the set $D$ of all declared types is finite
- there are no more than $|D|^2$ different equivalence queries
- repeated queries for the same inputs are automatically satisfied
\implies \text{termination is ensured}
Subtyping ≤

On the arithmetic basic types `char`, `int`, `long`, etc. there exists a rich subtype hierarchy.

Subtypes

- $t_1 \leq t_2$, means that the values of type $t_1$
  - form a subset of the values of type $t_2$;
  - can be converted into a value of type $t_2$;
  - fulfill the requirements of type $t_2$;
  - are assignable to variables of type $t_2$.

Example:
assign smaller type (fewer values) to larger type (more values)

```plaintext
int x;
double y;
y = x;
```

$t_1 \leq t_2 int \leq double$
Example: Subtyping

Extending the subtype relationship to more complex types, observe:

```c
string extractInfo( struct { string info; } x) {
    return x.info;
}
```

- we want `extractInfo` to be applicable to all argument structures that return a `string` typed field for accessor `info`
- the idea of subtyping on values is related to subclasses
- we use deduction rules to describe when \( t_1 \leq t_2 \) should hold...
Rules for Well-Typedness of Subtyping

\[ t \leq t' \]

\[ s \prec t \]

\[ \text{typedef } s \ A \]

\[
\begin{align*}
\text{struct} \{ s_1 \ a_1; \ldots \ s_j \ a_j; \} & \quad \text{struct} \{ t_1 \ a_1; \ldots \ t_k \ a_k; \} \\
\text{struct} \{ \text{int } u, \text{int } v \} & \quad x; \\
\text{struct} \{ \text{int } u \} & \quad y; \\
y & = x;
\end{align*}
\]
Rules and Examples for Subtyping

Examples:

\[
\begin{align*}
\text{struct } \{ \text{int } a; \text{ int } b; \} & \leq \text{ struct } \{ \text{float } a; \} \\
\text{int } (\text{int}) & \not\leq \text{ float } (\text{float}) \\
\text{int } (\text{float}) & \leq \text{ float } (\text{int})
\end{align*}
\]

Definition

Given two function types in subtype relation \( s_0(s_1, \ldots, s_n) \leq t_0(t_1, \ldots, t_n) \) then we have

- **co-variance** of the return type \( s_0 \leq t_0 \) and
- **contra-variance** of the arguments \( s_i \geq t_i \) für \( 1 < i \leq n \)
Check if $S_1 \leq R_1$:

$$R_1 = \text{struct \{int } a; R_1 (R_1) f; \}$$

$$S_1 = \text{struct \{int } a; \text{int } b; S_1 (S_1) f; \}$$

$$R_2 = \text{struct \{int } a; R_2 (S_2) f; \}$$

$$S_2 = \text{struct \{int } a; \text{int } b; S_2 (R_2) f; \}$$
Subtypes: Application of Rules (II)

Check if $S_2 \leq S_1$:

$$R_1 = \text{struct} \{ \text{int } a; \ R_1(R_1) \ f; \}$$

$$S_1 = \text{struct} \{ \text{int } a; \ \text{int } b; \ S_1(S_1) \ f; \}$$

$$R_2 = \text{struct} \{ \text{int } a; \ R_2(S_2) \ f; \}$$

$$S_2 = \text{struct} \{ \text{int } a; \ \text{int } b; \ S_2(R_2) \ f; \}$$
Subtypes: Application of Rules (III)

Check if $S_2 \leq R_1$:

\[
R_1 = \text{struct}\{\text{int } a; \ R_1(R_1) \ f; \}\n\]
\[
S_1 = \text{struct}\{\text{int } a; \ \text{int } b; \ S_1(S_1) \ f; \}\n\]
\[
R_2 = \text{struct}\{\text{int } a; \ R_2(S_2) \ f; \}\n\]
\[
S_2 = \text{struct}\{\text{int } a; \ \text{int } b; \ S_2(R_2) \ f; \}\n\]
for presentational purposes, proof trees are often abbreviated by omitting deductions within the tree

structural sub-types are very powerful and can be quite intricate to understand

Java generalizes structs to objects/classes where a sub-class $A$ inheriting form base class $O$ is a subtype $A \leq O$

subtype relations between classes must be explicitly declared