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);
}
```

- within the body of the loop, the definition of `A` is shadowed by the local definition
- each declaration of a variable `v` requires allocating memory for `v`
- accessing `v` requires finding the declaration the access is bound to
- a binding is not visible when a local declaration of the same name is in scope
Rapid Access: Replace Strings with Integers

Idea for Algorithm:
- **Input:** a sequence of strings
- **Output:** sequence of numbers
- Table that allows to retrieve the string that corresponds to a number

Apply this algorithm on each identifier during **scanning**.

Implementation approach:
- count the number of new-found identifiers in int \( \text{count} \)
- maintain a **hashtable** \( S : \text{String} \rightarrow \text{int} \) to remember numbers for known identifiers

We thus define the function:

```java
int indexForIdentifier(String w) {
    if (S(w) == undefined) {
        S = \{ w \rightarrow \text{count}; \}
        return \text{count}++;  
    } else return S(w);
}
```

Example: Replacing Strings with Integers

Input:

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

If Peter Piper picked a peck of pickled peppers
whereas the peck of pickled peppers Peter Piper picked

Output:

Implementation: Hashtables for Strings

- allocate an array \( M \) of sufficient size \( m \)
- 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 \((x) = (x_0, \ldots, x_{r-1})\):

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

for some prime number \( p \) (e.g., 31)

\( \times \) 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: \( O(1) \)

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
  - if we visit a declaration, we push it onto the stack of its identifier
    - 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: A Table of Stacks

```c
// Abstract locations in comments
int a, b; // V, W
b = 5;
if (b>3) {
    int a, c; // X, Y
    a = 3;
c = a + 1;
b = c;
} else {
    int c; // Z
c = a + 1;
b = c;
}
b = a + b;
```

Decl-Use Analysis: Annotating the Syntax Tree

```c
d declaration node
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

```
a
b
```

in front of if-statement

Type Definitions in C

A type definition is a synonym for a type expression. In C they are introduced using the typedef keyword. Type definitions are useful

- as abbreviation:

```
typedef struct { int x; int y; } point_t;
```

- to construct recursive types:

```
Possible declaration in C: more readable:
struct list {                     typedef struct list list_t;
    int info;
    struct list* next;             struct list {
}                                 int info;
                                 struct list* next;
}                                 };
struct list* head;               struct list* head;
```

208/288 209/288 210/288 211/288
Type Definitions in C

The C grammar distinguishes `typedef-name` and `identifier`. Consider the following declarations:

```c
typedef struct { int x, y } point_t;
point_t origin;
```

Relevant C grammar:
- `declaration` → `(declaration-specifier)+ declarator ;`
- `declaration-specifier` → `static | volatile ... typedef | void | char | char ... typename`
- `declarator` → `identifier | ...`

Type Definitions in C: Solutions

Solution is difficult:

Type Expressions

Types are given using type-`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

Semantic Analysis

Chapter 3: Type Checking
Type Checking

Problem:

Given: A set of type declarations $\Gamma = \{ \ell_1 x_1; \ldots \ell_m x_m; \}$
Check: Can an expression $e$ be given the type $\ell$?

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: $\Gamma \vdash e : t_c$  $(t_c$ type of constant $e$)
Var: $\Gamma \vdash x : \Gamma(x)$  $(x$ Variable)

Rules:

Ref: $\Gamma \vdash e : t$  $\Gamma \vdash & e : t *$
Deref: $\Gamma \vdash e : t *$  $\Gamma \vdash * e : t$

Type Checking using the Syntax Tree

Check the expression $\star a \{ f (b \rightarrow c) \} + 2$:

- 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 for C-like Languages

More rules for typing an expression:

Array:

$\Gamma \vdash e_1 : t$  $\Gamma \vdash e_2 : \text{int}$
$\Gamma \vdash e_1[e_2] : t$

Array:

$\Gamma \vdash e_1 : t[]$  $\Gamma \vdash e_2 : \text{int}$
$\Gamma \vdash e_1[e_2] : t$

Struct:

$\Gamma \vdash e : \text{struct} \{ t_1 a_1; \ldots ; t_m a_m; \}$

App:

$\Gamma \vdash e : t \{ t_1, \ldots , t_m \}$  $\Gamma \vdash e_1 : t_1$  $\ldots$  $\Gamma \vdash e_m : t_m$

Op $\Box$:

$\Gamma \vdash e_1 : t$  $\Gamma \vdash e_2 : t$
$\Gamma \vdash e_1 \Box e_2 : t$

Explicit Cast:

$\Gamma \vdash e : t_1$  $t_1$ can be converted to $t_2$
$\Gamma \vdash (t_2) e : t_2$
Example: Type Checking

Given expression \( \ast a[f(b\rightarrow c)] + 2 \) and

\[ \Gamma = \{
  \begin{array}{l}
  \text{struct list} \{ \text{int info}; \text{struct list} \ast \text{next}; \}; \\
  \text{int} f(\text{struct list} \ast l); \\
  \text{struct} \{ \text{struct list} \ast c; \} \ast b; \\
  \text{int} \ast a[l]; \\
  \end{array}
\]
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;
}
```

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

Consider declarations `struct list l` and `struct list1 l1`.

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:

```c
typedef A t
```

(we omit the \( T \)). Then define the following rules: