These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Project 1 and Quiz 2 have been graded

• Second project – Please continue working on it

• Sixth homework on Quiz 3 will be posted over the weekend
Context-Sensitive Analysis

EaC Chapter 4
ALSU Chapter 5
Types and Type Systems

**Type**: A set of values and meaningful operations on them

Types provide semantic “sanity checks” (consistency checks) and determine efficient implementations for data objects.

Types help identify:
- errors, if an operator is applied to an incompatible operand
  - dereferencing of a non-pointer
  - adding a function to something
  - incorrect number of parameters to a procedure
  - ...
- which operation to use for overloaded names and operators, or what type coercion to use (e.g.: 3.0 + 1)
- identification of polymorphic functions
Types and Type Systems

Type system: Each language construct (operator, expression, statement, ...) is associated with a type expression. The type system is a collection of rules for assigning type expressions to these constructs.

Type expressions for
- basic types: integer, char, real, boolean, typeError
- constructed types, e.g., one-dimensional arrays:
  array(lb, ub, elem_type), where elem_type is a type expression

A type checker implements a type system. It computes or "constructs" type expressions for each language construct.
Example type inference rule:

\[
E \vdash e_1 : \text{integer}, \ E \vdash e_2 : \text{integer} \\
\underline{E \vdash e_1 + e_2 : \text{integer}}
\]

where \( E \) is a type environment that maps constants and variables to their type expressions.

Questions: How to specify rules that allow type coercion (type widening) from integers to reals in arithmetic expressions?

\[
3.0 + 1 \quad \text{or} \quad 1 + 3.0
\]
Example type inference rule pointer dereferencing:

\[
\frac{E \vdash e : ???}{E \vdash *e : ???}
\]

where E is a type environment that maps constants and variables to their type expressions.
Example type inference rule pointer dereferencing:

\[
\begin{align*}
E \vdash e : \text{pointer(integer)} \\
\hline
E \vdash *e : \text{integer}
\end{align*}
\]

where \(E\) is a type environment that maps constants and variables to their type expressions.

\text{pointer(...) is now part of the type expression language such as array(...).}
Types and Type Systems

Example type inference rule pointer dereferencing:

\[ E \vdash e : \text{pointer}(\beta) \]
\[ \overline{\quad} \]
\[ E \vdash *e : \beta \]

where E is a type environment that maps constants and variables to their type expressions.

Type expressions may also contain type variables such as \( \beta \). Type variables can denote any type expression.

Type variables are needed to express polymorphic types.
Example type inference rule address computation:

\[
E \vdash e : ??? \\
\hline
E \vdash \&e : ???
\]

where E is a type environment that maps constants and variables to their type expressions.

What about a polymorphic version of this rule?
Example type inference rule address computation:

\[ \begin{align*}
E \vdash e : \text{integer} \\
\hline \\
E \vdash &e : \text{pointer(integer)}
\end{align*} \]

where \( E \) is a type environment that maps constants and variables to their type expressions.

What about a polymorphic version of this rule?
Example type inference rule address computation:

\[
\begin{align*}
  & \quad E \vdash e : \text{integer} \\
\hline
  \quad & \quad E \vdash \&e : \text{pointer(integer)}
\end{align*}
\]

where E is a type environment that maps constants and variables to their type expressions.

What about a polymorphic version of this rule?

\[
\begin{align*}
  & \quad E \vdash e : \beta \\
\hline
  \quad & \quad E \vdash \&e : \text{pointer(}\beta\text{)}
\end{align*}
\]
Formal proof that a program can be typed correctly.

```c
int a;
E = { a: integer }

...*(&a) + 3 ...
```
Formal proof that a program can be typed correctly.

```plaintext
int a;   E = { a: integer }

...  
...*(&a) + 3 ... 

E |- a: integer
   ---------------------
E |- (&a): pointer (integer)
   ---------------------------------
E |- *(*&a): integer , E |- 3: integer   E = { a: integer, 3: integer}
   ---------------------------------
E |- *(*&a) + 3 : integer
```
Type Names and Type Systems

Programmers may define their own types and give them names:

```plaintext
type my_int is int;
...
int a; a: integer
my_int b; b: my_int
...
... a + b ...
```

Type names can also be part of the type expression language.
Note: type names and type variables are different!
Type Equivalence

Structural -- type equivalence: type names are expanded
Name -- type equivalence: type names are not expanded

Example:

```pascal
type A is array(1..10) of integer;
type B is array(1..10) of integer;
a : A;
b : B;
c, d : array(1..10) of integer;
e : array(1..10) of integer;
```

Answer: structural equivalence:
name equivalence:
Type Equivalence

**Structural** -- type equivalence: type names are expanded

**Name** -- type equivalence: type names are not expanded

Example:

```plaintext
type A is array(1..10) of integer;
type B is array(1..10) of integer;
a : A;
b : B;
c, d: array(1..10) of integer;
e: array(1..10) of integer;
```

Answer: structural equivalence: (a, b, c, d, e)

name equivalence: (a); (b); (c, d, e);
Revisit our type inference rule for “+”.

```c
exp : exp '+' exp { if ($1 == TYPE_INT && $3 == TYPE_INT)
    $$ = TYPE_INT;
else {
    $$ = TYPE_INT;
    printf("\n***Error: illegal operand types\n");
}
}
```

PROJECT HINT: The definition of type expression as C types (structs) should be done in `attr.h`. `attr.c` may contain helper functions. The assignment of type expression C types to terminals and nonterminals of the grammar is done in `parse.y`. 
Lexically-scoped Symbol Tables

The problem
• The compiler needs a distinct record for each declaration
• Nested lexical scopes admit duplicate declarations

The interface
• insert(name, level) - creates record for name at level
• lookup(name, level) - returns pointer or index
• delete(level) - removes all names declared at level

Many implementation schemes have been proposed (see § B.4)
• We’ll stay at the conceptual level
• Hash table implementation is tricky, detailed, & fun

Symbol tables are compile-time structures the compiler use to resolve references to names. We’ll see the corresponding run-time structures that are used to establish addressability later.
Example

procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure r {
            int x, y, z
            ...
        }
    }
    procedure s {
        int x, a, v
        ...
    }
    ... r ... s
    ...
}
... q ...

B0: {
    int a, b, c
B1: {
    int v, b, x, w
B2: {
    int x, y, z
    ...
    }
B3: {
    int x, a, v
    ...
    }
    ...
    }
    ...
    }
    ...
Example

Picturing it as a series of Algol-like procedures
Lexically-scoped Symbol Tables

High-level idea
- Create a new table for each scope
- Chain them together for lookup

“Chain of tables” implementation
- `insert()` may need to create table
- it always inserts at current level
- `lookup()` walks chain of tables & returns first occurrence of name
- `delete()` throws away table for level `p`, if it is top table in the chain

Individual tables can be hash tables.
High-level idea

- Create a new table for each scope
- Chain them together for lookup

Remember

If we add the subscripts, the relationship between the code and the table becomes clear

The names visible in $s$

- $a_{11}$, $b_4$, $c_2$
- $v_{12}$, $w_6$, $x_{10}$
- no $y$ or $z$
### Stack organization

<table>
<thead>
<tr>
<th>nextFree</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>w</td>
</tr>
<tr>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s (level 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>q (level 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ρ (level 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

### Implementation

- **insert()** creates new level pointer if needed and inserts at nextFree
- **lookup()** searches linearly from nextFree-1 forward
- **delete()** sets nextFree to the equal the start location of the level deleted.

### Advantage
- Uses much less space

### Disadvantage
- Lookups can be expensive
Stack organization

Implementation
• insert () creates new level pointer if needed and inserts at nextFree
• lookup () searches linearly from nextFree-1 down stack
• delete () sets nextFree to the equal the start location of the level deleted.

Advantage
• Uses much less space

Disadvantage
• Lookups can be expensive
Threaded stack organization

Implementation
- **insert ()** puts new entry at the head of the list for the name
- **lookup ()** goes direct to location
- **delete ()** processes each element in level being deleted to remove from head of list

Advantage
- lookup is fast

Disadvantage
- delete takes time proportional to number of declared variables in level
Threaded stack organization

- Insert () puts new entry at the head of the list for the name
- Lookup () goes direct to location
- Delete () processes each element in level being deleted to remove from head of list

Advantage
- Lookup is fast

Disadvantage
- Delete takes time proportional to number of declared variables in level

Implementation

- $a_{11}$, $b_4$, $c_2$, $v_{12}$, $w_6$, $x_{10}$, no $y$ or $z$

$h(x)$

```
| v  | 12 |
| a  | 11 |
| x  | 10 |
| w  |  6 |
| x  |  5 |
| b  |  4 |
| v  |  3 |
| c  |  2 |
| b  |  1 |
| a  |  0 |
```
Things to do and next class

Work on the project!

**Code generation (EaC Chapter 7)**

**Optimization: CSE**