Compilers

Semantic Analysis & Symbol Table Management
Static versus Dynamic Checking

- **Static checking**: the compiler enforces programming language’s *static semantics*
  - Program properties that can be checked at compile time

- **Dynamic semantics**: checked at run time
  - Compiler generates verification code to enforce programming language’s dynamic semantics
Static checking examples:

- **Type checks**: in $A := B + C$, all operands should have the same type

- **Flow-of-control checks**: check whether a e.g. break statement has somewhere to return control.

- **Uniqueness checks**: In some languages names must be unique

- **Named-related checks**: In ADA for loops can have name, and it must appear twice (before the for keyword and before the end statement).
Type Checks, Overloading, and Coercion

```c
int op(int), op(float);
int f(float);
int a, c[10], d;

d = c+d;       // FAIL
*d = a;        // FAIL
a = op(d);     // OK: overloading (C++)
a = f(d);      // OK: coercion of d to float
```
Flow-of-Control Checks

```c
myfunc()
{
    ...
    break; // ERROR
}
```

```c
myfunc()
{
    ...
    while (n)
    {
        ...
        if (i>10)
            break; // OK
    }
}
```

```c
myfunc()
{
    ...
    switch (a)
    {
        case 0:
        {
            ...
            break; // OK
        }
        case 1:
        {
            ...
        }
    }
}
```
Uniqueness Checks

```c
myfunc()
{ int i, j, i; // ERROR
  ...
}
```

```c
struct myrec
{ int name;
  ...;
};
```

```c
cnufym(int a, int a) // ERROR
{ ...;
}
```

```c
struct myrec // ERROR
{ int id;
  ...;
};
```
LoopB: for (int J = 0; J < m; J++)
{
    LoopA: for (int I = 0; I < n; I++)
    {
        ...
        if (a[I] == 0)
            break LoopB; // Java labeled loop
        ...
    }
}
One-Pass versus Multi-Pass Static Checking

- **One-pass compiler**: static checking in C, Pascal, Fortran, and many other languages is performed in one pass while intermediate code is generated. (Influences design of a language: placement constraints)

- **Multi-pass compiler**: static checking in Ada, Java, and C# is performed in a separate phase, sometimes by traversing a syntax tree multiple times.
Dynamic checking:

- A piece of object code is added to the compiled program to perform the checking in the execution time

- Example:
  
  ```
  var a[10] int; ...; read (l); a(l) := 0;
  ```

- Generated code is as such the last statement would have been:

  ```
  If l <= 10 then a(l) := 0 else print ("subscript out of range error")
  ```
**OPERATIONS:**
Insert (string, token_ID)
Lookup (string)

**NOTICE:**
Reserved words are placed into symbol table for easy lookup.

Attributes may be associated with each entry, i.e., typing info, size info, memory address, etc.
program sort(input, output);
var a : array [0 .. 10] of integer; x : integer;

procedure readarray;
var i : integer;
begin ...
end;

procedure exchange(i, j, : integer);
begin
  x := a[i]; a[i] := a[j]; a[j] := x
end

procedure quicksort(m, n: integer);
var k, v : integer;

function partition(y, z: integer) : integer;
var i, j : integer;
begin ...
end { partition };
When entering readarray

<table>
<thead>
<tr>
<th>symbol table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lexeme</strong></td>
</tr>
<tr>
<td>sort</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>readarray</td>
</tr>
<tr>
<td>i</td>
</tr>
</tbody>
</table>
After entering quicksort

---

**Symbol Table Management**

<table>
<thead>
<tr>
<th>lexeme</th>
<th>type</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>readarray</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>exchange</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td>quicksort</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>k</td>
<td>int</td>
<td>6</td>
</tr>
<tr>
<td>v</td>
<td>int</td>
<td>7</td>
</tr>
</tbody>
</table>