Static Checking and Type Systems

Semantic Analysis

Static versus Dynamic Checking

- *Static checking*: the compiler enforces programming language's *static semantics*
  – Program properties that can be checked at compile time
- *Dynamic checking*: checked at run time
  – Compiler generates verification code to enforce programming language's dynamic semantics

Static Checking

- Typical examples of static checking are
  - Type checks
  - Flow-of-control checks
  - Uniqueness checks
  - Name-related checks

Type Checks, Overloading, Coercion, and Polymorphism

```
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

```
myfunc()
{ …
  break; // ERROR
}
```

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

```
myfunc()
{ …
  switch (a)
  { case 0:
      …
      break; // OK
    case 1:
      …
  }
}
```

Uniqueness Checks

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

```
cnufym(int a, int a) // ERROR
{    …
}
```

```
struct myrec
{ int name;
};
struct myrec // ERROR
{ int id;
};
```

Name-Related Checks

```
LoopA: for (int I = 0; I < n; I++)
       { …
         if (a[I] == 0)
           break LoopB; // Java labeled loop
         …
       }
```

One-Pass versus Multi-Pass Static Checking

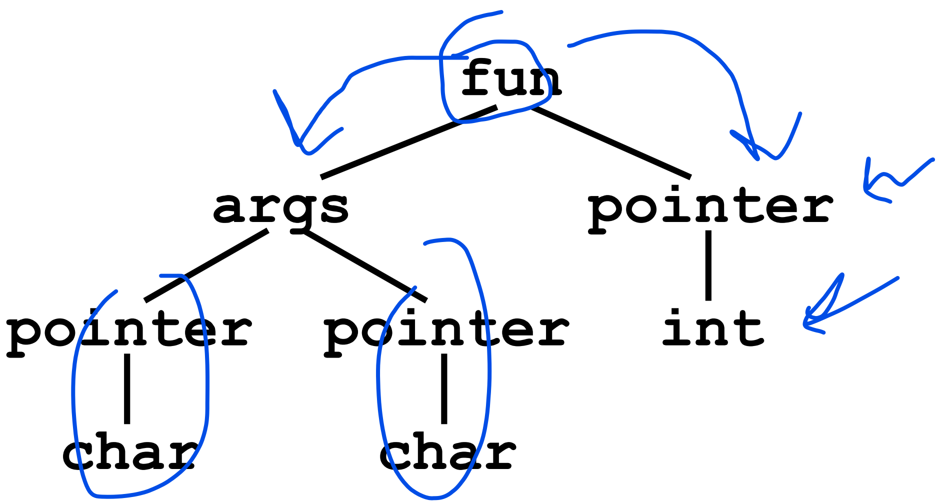
- *One-pass compiler*: static checking for C, Pascal, Fortran, and many other languages is performed in one pass while intermediate code is generated
  - Influences design of a language: placement constraints
- *Multi-pass compiler*: static checking for Ada, Java, and C# is performed in a separate phase, sometimes by traversing the syntax tree multiple times

Type Expressions

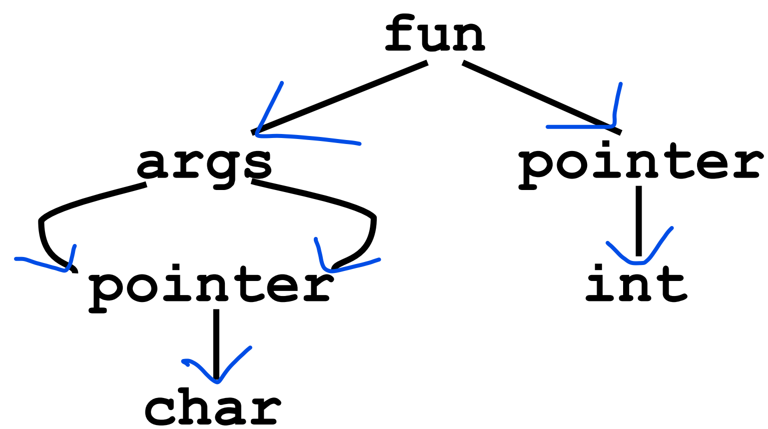
- *Type expressions* are used in declarations and type casts to define or refer to a type
  - *Primitive types*, such as **`int`**, **`bool`**,**`char`**, **`float`**, **`etc`**.
  - *Type constructors*, such as pointers, arrays, records, structures, classes, and functions.
  - *Type names*, such as typedef in C and named types in Pascal, refer to type expressions.

Graph Representations for Type Expressions

```
int *fun(char*,char*)
```



Tree forms



DAGs

Name Equivalence

- Each *type name* is a distinct type, even when the type expressions the names refer to are the same
- Types are identical only if names match

Used by Pascal

```
type link = ^node;
var next : link;
    last : link;
       p : ^node;
    q, r : ^node;
```

With name equivalence in Pascal:

$$\texttt{p} \neq \texttt{next}$$
$$\texttt{p} \neq \texttt{last}$$
$$\texttt{p} = \texttt{q} = \texttt{r}$$
$$\texttt{next} = \texttt{last}$$

Type Systems

- A ***type system*** defines a set of **types** and **rules** to assign types to various programming language constructs, such as variables, expressions, functions or modules. The rules are similar to Syntax Directed Definitions (SDDs).

Eg., "*if both operands of addition are of type integer, then the result is of type integer*" for following expression.

a = b + c

Simple Language Example: Declarations

$D \rightarrow$ **id** **:** T	{ *addtype*(**id**.entry, T.type) }
$T \rightarrow$ **boolean**	{ T.type := *boolean* }
$T \rightarrow$ **char**	{ T.type := *char* }
$T \rightarrow$ **integer**	{ T.type := *integer* }
$T \rightarrow$ **array [1..num] of** T_1	{ T.type := *array*(1..**num**.val, T_1.type) }
$T \rightarrow$ **^** T_1	{ T.type := *pointer*(T_1)

Parametric types:
type constructor

a,b: integer // int a,b;
b: array[1..20] of char /// char b[20];

Simple Language Example: Checking Expressions

$E \rightarrow E_1 + E_2$ { E.type := **if** E_1.type = *integer* **and** E_2.type = *integer* **then** *integer*
else **if** E_1.type = integer **and** E_2.type = float **then** *float*
else *type_error*
}

Simple Language Example: Checking Expressions (cont'd)

$E \rightarrow E_1$ **and** E_2 {

E.type :=

if E_1.type = *boolean* **and** E_2.type = *boolean*

then *boolean*

else *type_error*

}

Simple Language Example: Checking Expressions (cont'd)

$E \rightarrow E_1$ **[** E_2 **]** {
E.type := **if** E_2.type = *integer* **and** E_1.type = *array*(s, t)
then t
else *type_error*
}

// a = b[10]

Simple Language Example: Checking Expressions (cont'd)

$E \rightarrow E_1$ **^**

{
E.type := **if** E_1.type = *pointer*(t)
then t
else *type_error*
}

A Simple Language Example: Functions

$T \rightarrow T$ **->** T

$E \rightarrow E$ **(** E **)**

For example:

```
v : integer;
odd : integer -> boolean;
if odd(3) then
    v := 1;
```

Simple Language Example: Function Declarations

$$T \rightarrow T_1 \text{ -> } T_2 \quad \{ \; T.\text{type} := \mathit{function}(T_1.\text{type}, T_2.\text{type}) \; \}$$

Parametric type:
type constructor

Simple Language Example: Checking Function Invocations

$E \rightarrow E_1 \textbf{ (} E_2 \textbf{)}$ {

E.type :=

if E_1.type = *function*(s, t) **and** E_2.type = s

then t

else *type_error*

}