Type checking

Types help identify

- errors, if an operator is applied to an incompatible operand.
  - dereferencing of pointers only
  - adding a function name to something
  - the correct number of parameters to a procedure

- which operation to use for overloaded names and operators (polymorphism)
Type systems

Each operator and expression in a program has a type

**basic types:** integer, real, character, *etc.*

**constructed types:** arrays, records, sets, pointers, functions

A **type system** is a collection of rules for assigning *type expressions* to variables.

A **type checker** implements the type system.

Example type rules

- If both operands of the arithmetic operators of addition, subtraction, and multiplication are of type integer, then the result is of type integer. (Pascal definition)

- The result of the unary & operator is a pointer to the object referred to by the operand. If the operand is of type "foo", then the type of the result is a "pointer to foo". (C and C++ definition)
Type expressions

1. The type of a language construct.

2. A basic type is a type expression. A special basic type, typeError will signal an error. A basic type void denotes an untyped statement.

3. Since type expressions may be named, a type name is a type expression.

4. Type expressions may contain variables whose values are type expressions.

5. A type constructor applied to type expressions is a type expression. Examples:

(a) arrays
(b) products
(c) records
(d) pointers
(e) functions
Type constructors

Type constructions include the following:

- **Arrays**
  
  If $T$ is a type expression, then $\text{array}(I, T)$ is a type expression denoting the type of an array with elements of type $T$ and index set $I$. $I$ is often a range, e.g.,

  \[
  \text{var A: array}[1\ldots10]\text{ of integer}
  \]

  associates the type expression

  \[
  \text{array}(1\ldots10, \text{integer})
  \]

  with $A$

- **Products**
  
  If $T_1$ and $T_2$ are type expressions, then their cartesian product $T_1 \times T_2$ is a type expression.
Type constructors (cont.)

• Records

The difference between a record and a product is that the fields of a record have names. The record type constructor will be applied to a tuple formed from field names and field types. e.g.,

```plaintext
type row = record
    address: integer
    lexemem: array [1...15] of char
end
var table: array[1...101] of row
```

declares the type name row representing the type expression:

```
record((address × integer) ×
    (lexeme × array(1...15, char)))
```

and the variable table to be an array of records of this type
Type constructors (cont.)

• Pointers

If $T$ is a type expression, then $\text{pointer}(T)$ is a type expression denoting the type “pointer to an object of type $T$.

• Functions

Functions map elements of one set, the domain, into another set, the range.

E.g., Pascal’s mod maps a pair of integers, $\text{int} \times \text{int}$ into an integer, type $\text{int}$

$$\text{int} \times \text{int} \to \text{int}$$

Note that type constructors are recursive

Can construct types such as:

1. pointer to pointer to integer
2. pointer to array of integer
3. array of pointer to integer
4. array of record of pointer to integer
Type checking

- static
- dynamic

```plaintext
table: array[0..255] of char;
i: integer

table[i] cannot be guaranteed at compile time to fall in the range of 0 to 255
```

A *sound* type system eliminates the need for dynamic checking for type errors, because it determines statically that these errors cannot occur.

A *strongly typed* language guarantees that the compiler will accept only program that execute without type errors.
A simple type checker

Using a synthesized attribute grammar, we will describe a type checker for arrays, pointers, statements, and functions.

Grammar for source language:

\[
\begin{align*}
P & ::= \ D \ ; \ E \\
D & ::= \ D \ ; \ E \ | \ \text{id}: \ T \\
T & ::= \ \text{char} \ | \ \text{integer} \ | \ \text{array} \ [\text{num}] \ \text{of} \ T \ | \ \uparrow T \\
E & ::= \ \text{literal} \ | \ \text{num} \ | \ \text{id} \ | \ E \ \text{mod} \ E \ | \ E[E] \ | \ E \uparrow
\end{align*}
\]

- Basic types *char*, *integer*, *typeError*

- assume all arrays start at 1, e.g.,
  
  array [256] of char
  
  results in the type expression
  
  \( \text{array}(1 \ldots 256, \text{char}) \)

- \( \uparrow \) builds a pointer type, so \( \uparrow \text{integer} \)
  
  results in the type expression *pointer(integer)*
A simple type checker (cont.)

Partial attribute grammar for the type system

\[
\begin{align*}
D & ::= \text{id: } T & \{ \text{addtype(id.entry, T.type) } \} \\
T & ::= \text{char} & \{ T.type \leftarrow \text{char} \} \\
T & ::= \text{integer} & \{ T.type \leftarrow \text{integer} \} \\
T & ::= \uparrow T_1 & \{ T.type \leftarrow \text{pointer}(T_1.type) \} \\
T & ::= \text{array [num] of } T & \{ T.type \leftarrow \\
& & \text{array(1...num.val, T_1.type) } \}
\end{align*}
\]
A simple type checker (cont.)

Type checking of expressions

\[
E ::= \text{literal} \quad \{ E.\text{type} \leftarrow \text{char} \} \\
E ::= \text{num} \quad \{ E.\text{type} \leftarrow \text{integer} \} \\
E ::= \text{id} \quad \{ E.\text{type} \leftarrow \text{lookup(id.entry)} \} \\
E ::= E_1 \text{ mod } E_2 
\{ E.\text{type} \leftarrow \text{if } E_1.\text{type} = \text{integer} \text{ and } E_2.\text{type} = \text{integer} \text{ then integer} \\
\text{else } \text{typeError} \} \\
E ::= E_1[E_2] 
\{ E.\text{type} \leftarrow \text{if } E_2.\text{type} = \text{integer} \text{ and } E_1.\text{type} = \text{array}(s,t) \text{ then } t \\
\text{else } \text{typeError} \} \\
E ::= E_1 \uparrow 
\{ E.\text{type} \leftarrow \text{if } E_1.\text{type} = \text{pointer} \text{ then } t \\
\text{else } \text{typeError} \} 
\]
Type checking statements

Statements do not typically have values, therefore we assign them the type \textit{void}. If an error is detected within the statement, it gets type \textit{typeError}.

\begin{align*}
S & ::= \text{id} \leftarrow E & \{ & S.type \leftarrow \text{id.type} = E.type \\
& & \text{then} & \text{void} \\
& & \text{else} & \text{typeError} \} \\
S & ::= \text{if E then } S_1 & \{ & S.type \leftarrow \text{if } E.type = \text{boolean} \\
& & \text{then } S_1.type \\
& & \text{else} & \text{typeError} \} \\
S & ::= \text{while E do } S_1 & \{ & S.type \leftarrow \text{if } E.type = \text{boolean} \\
& & \text{then } S_1.type \\
& & \text{else} & \text{typeError} \} \\
S & ::= S_1 ; S_2 & \{ & S.type \leftarrow \text{if } S_1.type = \text{void} \\
& & \text{then} & \text{void} \\
& & \text{else} & \text{typeError} \} \\
\end{align*}
Type checking functions

We add two new productions to the grammar to represent function declarations and applications

\[ T ::= T \rightarrow T \]  
declaration
\[ E ::= E ( E ) \]  
application

To capture the argument and return type, we use

\[ T ::= T_1 \rightarrow T_2 \{ T.type \leftarrow (T_1.type \rightarrow T_2.type) \} \]
\[ E ::= E_1 ( E_2 ) \{ E.type \leftarrow \text{if } E_2.type = s \text{ and } E_1.type = s \rightarrow t \text{ then } t \text{ else typeError} \} \]
Type equivalence

- easy for basic types, \textit{e.g.},
  integer is equivalent to integer

- passing arrays to procedures
  may not want to include array bounds

- \textit{structural equivalence} can be used to test equivalence, if we represent types as \textit{dags} taken from the typed parse tree
Type equivalence algorithm

function sequiv (s,t): boolean;
begin
    if s and t are the same basic type then
        return true
    else if s = array (s_1, s_2) and array (t_1, t_2) then
        return sequiv (s_1, t_1) and sequiv (s_2, t_2)
    else if s = s_1 × s_2 and and t = t_1 × t_2 then
        return sequiv (s_1, t_1) and sequiv (s_2, t_2)
    else if s = pointer (s_1) and t = pointer (t_1) then
        return sequiv (s_1, t_1)
    else if s = s_1 → s_2 and t = t_1 → t_2 then
        return sequiv (s_1, t_1) and sequiv (s_2, t_2)
    else
        return false
end