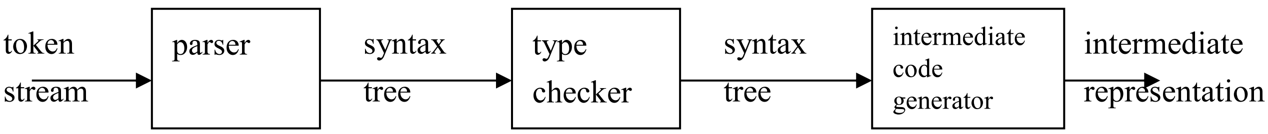
Type checking

Type 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 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.,

  var A: array [1..10] of integer

  associates the type expression

  array(1..10, 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.,

```
type row = record
        address: integer
        lexeme: 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 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, range.

  E.g., Pascal’s mod maps a pair of integers, int × int into an integer, type int

  $$\text{int} \times \text{int} \rightarrow \text{int}$$

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 interger
4. array of record of pointer to integer

Type checking

- static
- dynamic

    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:

P ::= D; E

D ::= D; E | id: T

T ::= char | integer | array [num] of T | ↑T

E ::= literal | num | id | E mod E | E[E] | E↑

- Basic types char, integer, typeError
- assume all arrays start at 1, e.g.,

  array [256] of char

  results in the type expression

  array(1..256, char)
- ↑ builds a pointer type, so ↑integer

  results in the type expression pointer(integer)

A simple type checker(cont.)

Partial attribute grammar for the type system

D ::= id: T	{addtype(id.entry, T.type)}
T ::= char	{T.type $\leftarrow$ char}
T ::= integer	{ T.type $\leftarrow$ integer}
T ::= $\uparrow T_1$	{ T.type $\leftarrow$ pointer(T_1.type)}
T ::= array [num] of T	{ T.type $\leftarrow$ array(1..num, T_1.type)}

A simple type checker(cont.)

Type checking of expressions

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

Type checking statements

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

$S ::= id \leftarrow E$	{ S.type $\leftarrow$ if id.type = E.type then void else typeError}
$S ::= \text{if } E \text{ then } S_1$	{ S.type $\leftarrow$ if E.type = boolean then S_1.type else typeError}
$S ::= \text{while } E \text{ do } S_1$	{ S.type $\leftarrow$ if E.type = boolean then S_1.type else typeError}
$S ::= S_1 ; S_2$	{ S.type $\leftarrow$ if S_1.type = void then void else typeError}

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

$T ::= T_1 \rightarrow T_2$ { $T.type \leftarrow (T_1.type \rightarrow T_2.type)$ }

$E ::= E_1 (E_2)$ {$E.type \leftarrow$ if $E_1.type = s \rightarrow t$
and $E_2.type = s$ then t
else typeError }

Type equivalence

- easy for basic types, e.g.,

  integer is equivalent to integer
- passing arrays to procedures

  may not want to include array bounds

- structural equivalence can be used to test equivalence, if we represent types as dags taken form 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(s1, s2) and t = array (t1, t2) then
        return sequiv(s1, t1) and sequiv (s2, t2)
    else if s = s1 × s2 and t = t1 × t2 then
        return sequiv(s1, t1) and sequiv (s2, t2)
    else if s = pointer(s1) and t = pointer(t1) then
        return sequiv(s1, t1)
    else if s = s1 → s2 and t = t1 → t2 then
        return sequiv(s1, t1) and sequiv (s2, t2)
    else
        return false
end
```