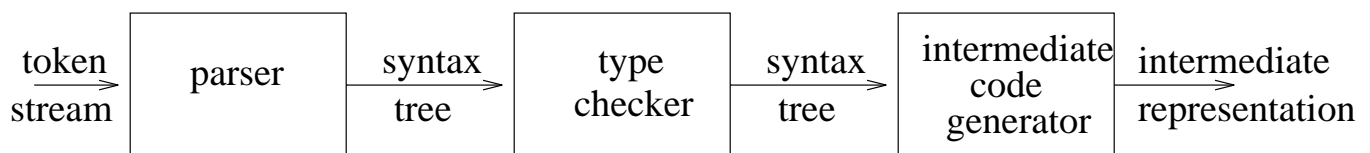


Type checking

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 $array(I, T)$ 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
  lexemem: array [1..15] of char
end
var table: array[1..101] of row
```

declares the type name *row* representing the type expression:

$$\text{record}((\text{address} \times \text{integer}) \times (\text{lexeme} \times \text{array}(1..15, \text{char})))$$

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, the range.

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

$$int \times int \rightarrow int$$

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

```
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:

$P ::= D ; E$

$D ::= D ; E \mid \text{id} : T$

$T ::= \text{char} \mid \text{integer} \mid \text{array} [\text{num}] \text{ of } T \mid \uparrow T$

$E ::= \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E[E] \mid E \uparrow$

- 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)
- \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

D ::= id: T	{ <i>addtype</i> (<i>id.entry</i> , <i>T.type</i>) }
T ::= char	{ <i>T.type</i> ← <i>char</i> }
T ::= integer	{ <i>T.type</i> ← <i>integer</i> }
T ::= ↑ T ₁	{ <i>T.type</i> ← <i>pointer</i> (<i>T_{1.type}</i>) }
T ::= array [num] of T	{ <i>T.type</i> ← <i>array</i> (<i>1..num.val</i> , <i>T_{1.type}</i>) }

A simple type checker (cont.)

Type checking of expressions

$E ::= \text{literal} \quad \{ E.type \leftarrow \text{char} \}$

$E ::= \text{num} \quad \{ E.type \leftarrow \text{integer} \}$

$E ::= \text{id} \quad \{ E.type \leftarrow \text{lookup}(\text{id.entry}) \}$

$E ::= E_1 \text{ mod } E_2 \quad \{ E.type \leftarrow \text{if } E_1.type = \text{integer} \text{ and } E_2.type = \text{integer} \text{ then } \text{integer} \text{ else } \text{typeError} \}$

$E ::= E_1[E_2] \quad \{ E.type \leftarrow \text{if } E_2.type = \text{integer} \text{ and } E_1.type = \text{array}(s,t) \text{ then } t \text{ else } \text{typeError} \}$

$E ::= E_1 \uparrow \quad \{ E.type \leftarrow \text{if } E_1.type = \text{pointer} \text{ then } t \text{ else } \text{typeError} \}$

Type checking statements

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

$$S ::= \text{id} \leftarrow E \quad \left\{ \begin{array}{l} S.type \leftarrow \text{if } \text{id}.type = E.type \\ \text{then } \text{void} \\ \text{else } \text{TypeError} \end{array} \right\}$$
$$S ::= \text{if } E \text{ then } S_1 \quad \left\{ \begin{array}{l} S.type \leftarrow \text{if } E.type = \text{boolean} \\ \text{then } S_1.type \\ \text{else } \text{TypeError} \end{array} \right\}$$
$$S ::= \text{while } E \text{ do } S_1 \quad \left\{ \begin{array}{l} S.type \leftarrow \text{if } E.type = \text{boolean} \\ \text{then } S_1.type \\ \text{else } \text{TypeError} \end{array} \right\}$$
$$S ::= S_1 ; S_2 \quad \left\{ \begin{array}{l} S.type \leftarrow \text{if } S_1.type = \text{void} \\ \text{then } \text{void} \\ \text{else } \text{TypeError} \end{array} \right\}$$

Type checking functions

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

$T ::= T \text{ “}\rightarrow\text{” } T$ declaration
 $E ::= E (E)$ application

To capture the argument and return type, we use

$T ::= T_1 \text{ ‘}\rightarrow\text{’ } T_2 \{ T.type \leftarrow (T_1.type \rightarrow T_2.type) \}$
 $E ::= E_1 (E_2) \{ E.type \leftarrow \text{if } E_2.type = s \text{ and } E_1.type = s \rightarrow t \text{ then } t \text{ else } typeError \}$

Type equivalence

- easy for basic types, *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
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 (s1, s2) and 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
```