Type Systems

Semantic Analysis

What can with do with semantic information for identifier x

- What kind of value is stored in x?
- How big is x?
- Who is responsible for allocating space for x?
- Who is responsible for initializing x?
- How long must the value of x be kept?
- If x is a procedure, what kinds of arguments does it take and
- What kind of return value does it have?

- Storage layout for local names

Introduction

- A source program should follow both the syntactic and semantic rules of the source language.
- Some rules can be checked statically during compile time and other rules can only be checked dynamically during runtime.
- Static checking includes the syntax checks performed by the parser and semantic checks such as type checks, flow-of control checks, uniqueness checks, and name-related checks.
- Here we focus on type checking.

Type Checking

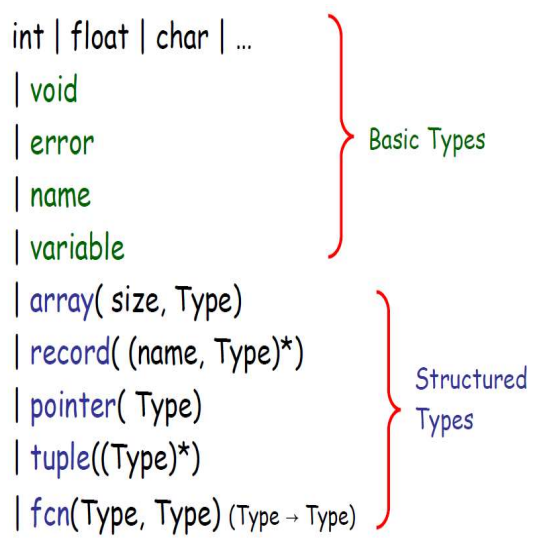
- Problem: Verify that a type of a construct matches that expected by its context.
- Examples:
  - mod requires integer operands (PASCAL)
  - * (dereferencing) – applied to a pointer
  - a[i] – indexing applied to an array
  - f(a1, a2, …, an) – function applied to correct arguments.
- Information gathered by a type checker:
  - Needed during code generation.

Use of Type

- Virtually all high-level programming languages associate types with values.
- Types often provide an implicit context for operations.
  - In C the expression x + y will use integer addition if x and y are int's, and floating-point addition if x and y are float's.
- Types can catch programming errors at compile time by making sure operators are applied to semantically valid operands.
  - For example, a Java compiler will report an error if x and y are String's in the expression x * y

Types

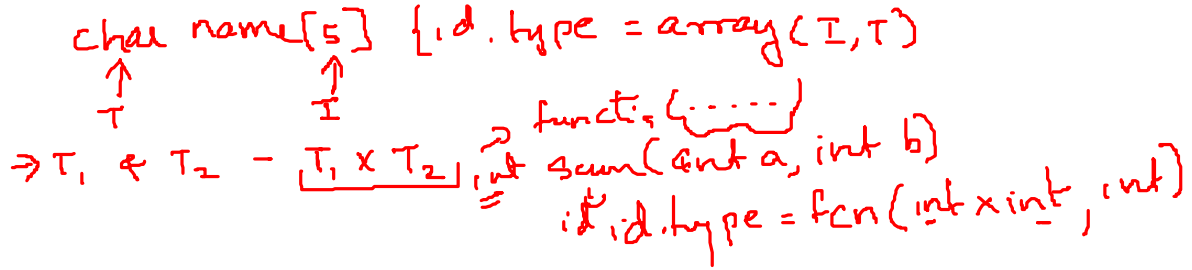
- Basic types are atomic types that have no internal structure as far as the programmer is concerned.
  - They include types like integer, real, boolean, and character.
  - Subrange types like 1..10 in Pascal and enumerated types like (violet, indigo, blue, green, yellow, orange, red) are also basic types.
- Constructed types include arrays, records, sets, and structures constructed from the basic types and/or other constructed types.
  - Pointers and functions are also constructed types.



Type Expressions

- **Type Expressions** denote the type of a language construct.
  - It is either a basic type or formed from other type expressions by applying an operator called a **type constructor**.
    - Example: a function from an integer to an integer.
    - A type constructor applied to a type expression is a type expression.
- Here we use type expressions formed from the following rules:
  - A basic type is a type expression. Other basic type expressions are **type-error** to signal the presence of a type error and **void** to signal the absence of a value.
  - If a type expression has a name then the name is also a type expression.

Type Constructors

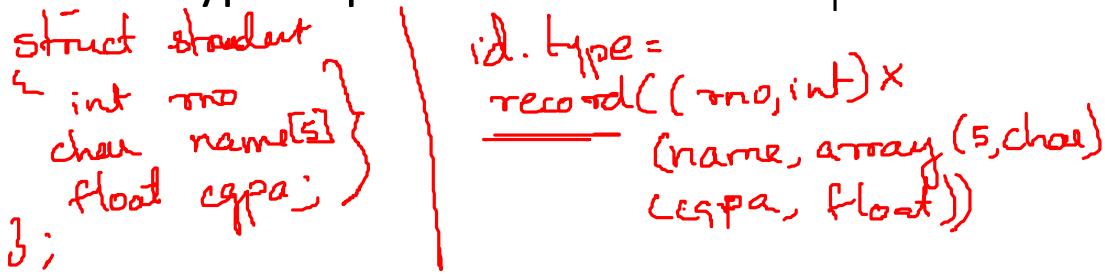


- Arrays. If **T** is a type expression and **I** is the type expression of an index set then ***array (I, T)*** denotes an *array of elements of type T.*
- Products. If ***T1*** and ***T2*** are type expressions, then their Cartesian product, ***T1 x T2*** , is a type expression.
  - For example if the arguments of a function are two reals followed by an integer then the type expression for the arguments is: **real x real x integer**.
- Records. The fields in a record (or structure) have names which should be included in the type expression of the record. The type expression of a record with n fields is:

  record (F_1 x F_2 x ... x F_n)

where if the *name of field i* is $name_i$ and the type expression of field i is T_i then F_i is:

$(name_i \times T_i)$.



Type Constructors

int *a; // id.type = pointer(int)
char *c; // id.type = pointer(char)

- *Pointers*.
  - If T is a type expression then ***pointer (T)*** denotes a pointer to an object of type T.
- Functions.
  - A function maps elements from its domain to its range.
  - The type expression for a function is: **D --> R** where

    D is the type expression for the domain of the function and R is the type expression for the range of the function.
  - For example, the type expression of the **mod operator** in Pascal is:

    integer x integer --> integer

    because it divides an integer by an integer and returns the integer remainder.

    The type expression for the domain of a function with no arguments is void and the type expression for the range of a function with no returned value is void:

    e.g., void --> void

    is the type expression for a procedure with no arguments and no returned value.

(fcn(D, R)

A Simple Typed Language

Program → Declaration; Statement

Declaration → Declaration; Declaration

| id: Type

Statement → Statement; Statement

| id := Expression

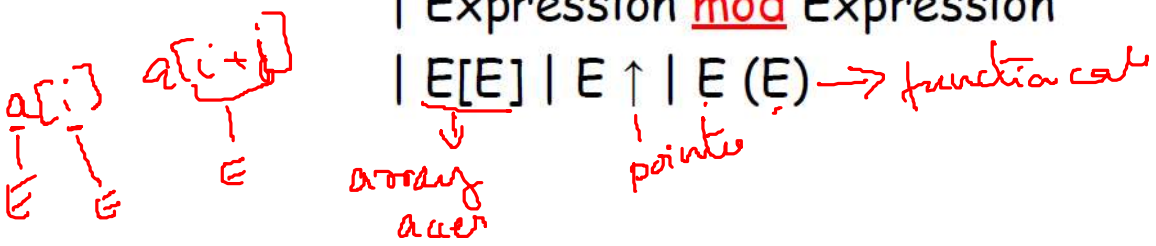
| if Expression then Statement

| while Expression do Statement

Expression → literal | num | id

| Expression mod Expression

| E[E] | E ↑ | E (E)



Type Checking Expressions

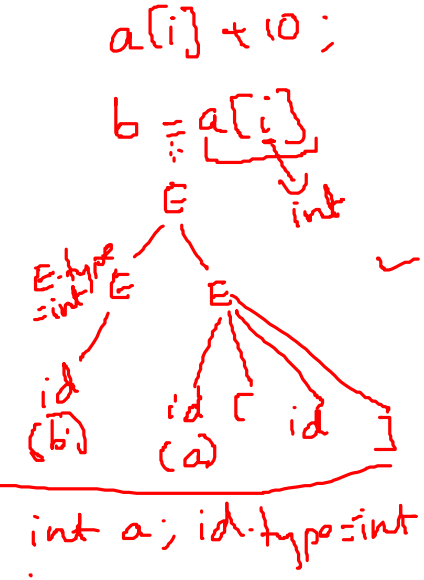
$E \rightarrow$ int_const { E.type = int }

$E \rightarrow$ float_const { E.type = float }

$E \rightarrow$ id { E.type = sym_lookup(id.entry, type) }

$E \rightarrow E_1 + E_2$ {E.type = if E_1.type $\notin$ {int, float} |
E_2.type $\notin$ {int, float})
then error
else if E_1.type == E_2.type == int
then int
else float }

Type Checking Expressions



$E \rightarrow E_1\,[E_2]$ {E.type = if E_1.type = array(S, T) ∧ E_2.type = int then T else error}

$E \rightarrow *E_1$ {E.type = if E_1.type = pointer(T) then T else error}

$E \rightarrow \&E_1$ {E.type = pointer(E_1.type)}

$E \rightarrow E_1(E_2)$ {E.type = if (E_1.type = fcn(S, T) ∧ E_2.type = S, then T else error}

int a[10]

type

arr

= stud [i]

int *b ; id ty pointer(int)

int a

a.type = int

→ a = *b

*E1

*b

value

integ

char *c ; pointer (char)

= *c + . . .

p = &a

S → parameter type

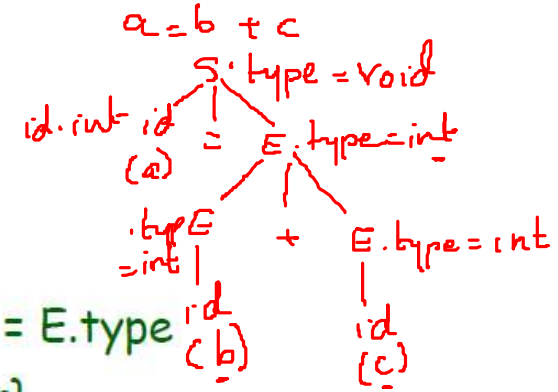
T → return type

(int x char x float) → void

sum(a, b)

T

Type Checking Statements



void, error

$S \rightarrow id := E$ {S.type := if id.type = E.type then void else error}

$S \rightarrow$ if E then S_1 {S.type := if E.type = boolean then S1.type else error}

$S \rightarrow$ while E do S_1 {S.type := if E.type = boolean then S_1.type} else error}

$S \rightarrow S_1; S_2$ {S.type := if S_1.type = void $\wedge$ S_2.type = void then void else error}