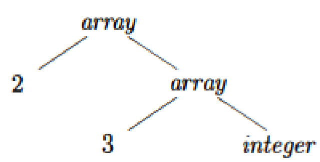
TYPES AND DECLARATIONS

- Type checking uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator.
- Translation Applications. From the type of a name, a compiler can determine the storage that will be needed for that name at run time.

Type Expressions:

- Types have structure, represented using type expressions: a type expression is either a basic type or is formed by applying an operator called a type constructor to a type expression. The sets of basic types and constructors depend on the language to be checked.
- The array type int[2][3] can be read as "array of 2 arrays of 3 integers each" and written as a type expression array(2, array(3, integer)). This type is represented by the tree. The operator array takes two parameters, a number and a type.



- A basic type is a type expression. Typical basic types for a language include boolean, char, integer, float, and void (denotes "the absence of a value.")
- A type name is a type expression.
- A type expression can be formed by applying the array type constructor to a number and a type expression.
- A record is a data structure with named fields. A type expression can be formed by applying the record type constructor to the field names and their types. Record types will be implemented by applying the constructor record to a symbol table containing entries for the fields.
- A type expression can be formed by using the type constructor → for function types. We write s→t for "function from type s to type t" Function types will be useful when type checking.
- If s and t are type expressions, then their Cartesian product s x t is a type expression. Products are introduced for completeness; they can be used to represent a list or tuple of types (e.g., for function parameters). We assume that x associates to the left and that it has higher precedence than→.
- Type expressions may contain variables whose values are type expressions.

Type Equivalence:

- When are two type expressions equivalent? Many type-checking rules have the form, "if two type expressions are equal then return a certain type else error."
- Potential ambiguities arise when names are given to type expressions and the names are then used in subsequent type expressions.
- The key issue is whether a name in a type expression stands for itself or whether it is an abbreviation for another type expression.

DECLARATIONS:

A simplified grammar that declares just one name at a time

$$\begin{array}{lcl} D & \rightarrow & T\ \textbf{id}\ ;\ D \mid \epsilon \\ T & \rightarrow & B\ C \mid \textbf{record}\ '\{'\ D\ '\}' \\ B & \rightarrow & \textbf{int} \mid \textbf{float} \\ C & \rightarrow & \epsilon \mid [\ \textbf{num}\ ]\ C \end{array}$$

- The above grammar deals with basic and array types. Nonterminal D generates a sequence of declarations. Nonterminal T generates basic, array, or record types.
- Nonterminal B generates one of the basic types int and float. Nonterminal C, for "component," generates strings of zero or more integers, each integer surrounded by brackets.
- An array type consists of a basic type specified by B, followed by array components specified by nonterminal C.
- A record type (the second production for T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

Sequences of Declarations:

- Languages such as C and Java allow all the declarations in a single procedure to be processed as a group. A variable offset, is used to keep track of the next available relative address.

$$\begin{array}{lll} P \rightarrow & & \{\ offset = 0;\ \} \\ & D & \\ D \rightarrow & T\ \textbf{id}\ ; & \{\ top.put(\textbf{id}.lexeme,\ T.type,\ offset); \\ & & \quad offset\ =\ offset + T.width;\ \} \\ & D_1 & \\ D \rightarrow & \epsilon & \end{array}$$

- The translation scheme deals with a sequence of declarations of the form T id, where T generates a type. Before the first declaration is considered, offset is set to 0.
- As each new name x is seen, x is entered into the symbol table with its relative address set to the current value of offset, which is then incremented by the width of the type of x.
- The semantic action within the production D → T id; D1 creates a symbol table entry by executing top.put(id.lexeme, T.type, offset). Here top denotes the current symbol table.
- The method top.put creates a symbol table entry for id.lexeme, with type T.type and relative address offset in its data area.

TRANSLATION OF EXPRESSIONS

- An expression with more than one operator, like a+b*c, will translate into instructions with at most one operator per instruction.
- An array references A[i][j] will expand into a sequence of three-address instructions that calculate an address for the reference.

Operations with Expressions:

- The syntax-directed definition builds up the three-address code for an assignment statement S using attribute code for S and attributes addr and code for an expression E.

- Attributes S.code and E.code denote the three-address code for S and E, respectively. Attribute E.addr denotes the address that will hold the value of E.

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \parallel gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \parallel E_2.code \parallel gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \parallel gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

- The last production E→id has the semantic rule which defines E.addr to point to the symbol-table entry for this instance of id. Let top denote the current symbol table.
- Function top.get retrieves the entry when it is applied to the string representation id.lexeme of this instance of id. E.code is set to the empty string.
- The semantic rules for E→El + E2 , generate code to compute the value of E from the values of El and E2. Values are computed into newly generated temporary names.
- If El is computed into E1.addr and E2 into E2.addr, then El + E2 translates into t= E1.addr + E2.addr, where t is a new temporary name. E. addr is set to t. A sequence of distinct temporary names t1, t2 , . . . is created by successively executing new Temp( ).
- E.code is built by concatenating E1.code, E2.code and an instruction that adds the values of E1 and E2.