# CS3005D Compiler Design

Winter 2024

Lecture #23

Symbol Table, Type Expressions

Saleena N
CSED NIT Calicut

February 2024

# Semantic Analysis

Semantic Analysis: checks the source program for semantic consistency with the language definition

- type checking - checks if each operator is applied to the right type of operands
- type conversions - coercions
- checks that can not be done during syntax analysis (e.g. whether a variable is declared before use)
- uses intermediate representation (e.g. abstract syntax tree) and information in the Symbol Table

# Symbol Tables

- data structure to maintain information regarding the *symbols*(names) in the source program - names of variables, symbolic constants, labels, procedure/function names ...
- one entry for each name, with values of its attributes
- information collected during initial phases of *Analysis*, used in later stages of compilation

# Symbol Table

- Possible attributes for names:
  - variable name - name (lexeme), type, scope, binding (relative position in storage)
  - function name - name, number of arguments, name and type of each argument, the method of passing each argument, return type
- Implementation - implementation of dictionary ADT (operations-insert, lookup and delete) - list, search trees, hash tables (commonly used for efficiency reasons)

# Scope

Scope of a declaration of a variable $x$ is the portion of the program in which uses of $x$ refer to this declaration

- Static scope / Lexical Scope - possible to determine the scope of a declaration statically - C, Java use static scope
- Dynamic scope - Scope can change dynamically(during runtime)

# Scope: block structured Languages

A block is a grouping of declarations and statements

- The scope of a declaration of name $x$ in a block $B$ is all of $B$ except for any blocks nested within $B$ in which $x$ is redeclared.
- Nested blocks - identifier $x$ is in the scope of the most-closely nested declaration of $x$

# Scope: example - C code fragment

```
int x; //global x
...
void f(...){
   int x; //local x
   ...
   x=3; //use of local x
   ...
}
...
x=x+1; //use of global x
```

# Block structured Languages: Symbol Table

- Maintain separate Symbol Tables - each block has its own symbol table with an entry for each declaration in that block
- Table for a nested block points to the table for its enclosing block (Chaining Symbol Tables)
- *lookup* by searching the chain of symbol tables, starting with the table for the current block
- Chaining results in a tree structure, since more than one block can be nested inside an enclosing block

# Type Expression

A *type expression* is used to denote the type of a language construct. A type expression can be

- a basic type e.g. *integer, boolean, char, float, void* (denotes absence of value), *type_error* (to signal error during type checking)
- a type name
- a type constructor applied to appropriate arguments

Type expression may contain variables whose values are type expressions.

# Type Expression: Type Constructor

- $array(I, t)$: the type constructor *array* applied to a number $I$ and a type expression $t$
- $s \rightarrow t$: the type constructor $\rightarrow$ applied to type expressions $s$ and $t$, denoting the type of functions with argument type $s$ and return type $t$
- $s \times t$: the type constructor $\times$ applied to type expressions $s$ and $t$, denoting list/tuple of types
- $record((l_1 \times t_1) \times (l_2 \times t_2))$: the type constructor *record* applied to a list of *(label, type)* pairs, denoting the type of a record
- *pointer(t)*: the type constructor *pointer* applied to a type $t$, denoting the type of a pointer pointing to an object of type $t$

# Type Expression : examples

- array(10, int): an array of size 10, base type is *int*
- $int \rightarrow boolean$: a function that takes an *int* type argument and returns a *boolean* value
- $int \times float \rightarrow boolean$: a function that takes two arguments (an *int* and a *float*) and returns a *boolean* value

# Type Equivalence

*Structural equivalence* of type expressions: the expressions are either the same basic type or are formed by applying the same type constructor to structurally equivalent types.

# References

**References**:

- Aho A.V., Lam M.S., Sethi R., and Ullman J.D. Compilers: Principles, Techniques, and Tools (ALSU). Pearson Education, 2007[1].

**Further reading**:

- ALSU Chapter2 - Sections 2.7 - 2.7.1, Chapter 6 - Section 6.3 - 6.3.1, 6.3.2

[1] some of the topics under type expressions are from another edition by Aho, Sethi and Ullman