CS 4031
Compiler Construction
Lecture 10

Mahzaib Younas

Lecturer, Department of Computer Science

FAST NUCES CFD

Types of SDT

- There is one more type called the L-attribute definition. The difference between S-attributed and L-attributed is as follows:

S-attributed grammar	L-attributed Grammar
1. It uses only synthesized attributes.	1. It allows both types. But if an inherited attribute is present, there is a restriction. The restriction is that each inherited attribute is restricted to inherit either from parent or from left sibling only.
2. Semantic actions can be placed only at the end of the right hand side of a production.	2. Semantic actions can be placed anywhere on the right hand side.
3. Attributes are generally evaluated during bottom-up parsing	3. Attributes are generally evaluated by traversing the parse tree depth first and left to right.

L-Attributed Grammar

- It allows both types, that is, synthesized as well as inherited. But if at all an inherited attribute is present, there is a restriction.
- The restriction is that each inherited attribute is restricted to inherit either from parent or from left sibling only.

Example:

- For example, consider the rule

$$A \rightarrow XYZPQ$$

assume that there is an inherited attribute. “i” is present with each nonterminal.

Valid Action of L Attributed Grammar	Invalid Action using L Attributed Grammar
Z•I = f(A•i|X•i|Y•i)	Z•i = f(P•i|Q•i)

Example

- Consider the following SDT:
- $A \rightarrow LM$ {L•i = f(A•i);
  M•i = f(L•s)
  A•s = f(M•s);}
- $A \rightarrow QR$ {R•i = f(A•i);
  Q•i = f(R•s)
  A•s = f(Q•s);}

Is it L-Attributed or S-Attributed?

Solution:

- Look at semantic actions. Here by looking at semantic actions we understand that “s” is synthesized and “i” is inherited. So actions in first rule are correct since “i” is dependent on parent or left siblings only. But look at the second action in A → QR, Q.i = f(R.s); this violates the basic property of L-attributed definition. Inherited attribute can inherit either from

Semantic Analyzer

A semantic analyzer mainly performs static checking. Static checks can be any one of the following type of checks:

1. Uniqueness checks
2. Flow of control checks
3. Type checks
4. Name-related checks

Uniqueness Check

- This ensures uniqueness of variables/objects in situations where it is required.
- Example
- in most of the languages no identifier can be used for two different definitions in the same scope.

Flow of Control Check

- Statements that cause flow of control to leave a construct should have a place to transfer flow of control. If this place is missing, it is confusion.
- Example
  - in C language, “break” causes flow of control to exit from the innermost loop. If it is used without a loop, it confuses where to leave the flow of control

Type Checks

- A compiler should report an error if an operator is applied to incompatible operands.
- Example
  - for binary addition, operands are array and a function is incompatible. In a function, the number of arguments should match with the number of formals and the corresponding types.

Name-Related Checks

- Sometimes, the same name must appear two or more times.
- Example
  - in ADA, a loop or a block may have a name that appears at the beginning and end of the construct. The compiler must check whether the same name is used at both places.

TYPE CHECKING

Type Checking

- A type is a set of values and operations on those values
- A language's type system specifies which operations are valid for a type
- The aim of type checking is to ensure that operations are used on the variable/expressions of the correct types

Type System

- Languages can be divided into three categories with respect to the type: –
- “untyped”
  - No type checking needs to be done
  - Assembly languages
- Statically typed
  - All type checking is done at compile time
  - Also, called strongly typed
- Dynamically typed
  - Type checking is done at run time
  - Mostly functional languages like Lisp, Scheme etc.

Type System

- A type system is a collection of rules for assigning type expressions to various parts of a program
- Different type systems may be used by different compilers for the same language
- In Pascal type of an array includes the index set. Therefore, a function with an array parameter can only be applied to arrays with that index set
- Many Pascal compilers allow index set to be left unspecified when an array is passed as a parameter

Types of Expression

1. Basic Type
2. Constrictor Type
3. Product Type
4. Record Types
5. Pointer
6. Function

Basic Type

- Basic Type
  - Boolean
  - Char
  - Integer
  - Real
  - Void
  - type_error

Constructor Type

- A type constructor applied to type expression is a type expression

  Array : array(I , T)

  - I is used to set the index.
  - T is used to declare the type of expression.

- Example:

  var C : array[1…20] of integer

  Array(1…20, integer)

Pointer

- Pointer (T) is a type expression denoting the type “pointer to an object of type T where T is a type of expression.
- Example:

Var ptr : *integer

  - Declares variable “ptr” to have type pointer (integer).

Function D →R

- a function is a mapping from elements of one set called domain to another set called range. We may treat functions in programming languages as map ping a domain type “Dom” to a range type “Rg.”. The type of such a function will be denoted by the type expression Dom
- Example
  - the built-in function mod, i.e. modulus of langauge has type expression

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

- Example:
  - The function having character in domain and pointer integer

$$\text{char} \times \text{char} \rightarrow \text{pointer(integer)}$$

Example:

1. Write type expression for a pointer to array of real, where the array index ranges from 1 to 100.
2. Write a type expression for a function whose domains are from integers to character and whose ranges are pointer to integer.

Solution:

Write type expression for a pointer to array of real, where the array index ranges from 1 to 100.

Solution: The type expression is pointer(array[1…100,real]).

Solution:

- Write a type expression for a function whose domains are from integers to character and whose ranges are pointer to integer.

Type expression is

Domain type expression is integer → character

Range type expression is pointer (integer)

The final type expression is (integer → character) → (pointer(integer))

Rules for Symbol Table Entry

$D \rightarrow id : T$	addtype(id.entry, T.type)
$T \rightarrow char$	T.type = char
$T \rightarrow integer$	T.type = int
$T \rightarrow T_1 *$	T.type = pointer(T_1.type)
$T \rightarrow T_1 [num]$	T.type = array(0..num-1, T_1.type)

Type Checking for Expression

$E \rightarrow$ literal	E.type = char
$E \rightarrow$ num	E.type = integer
$E \rightarrow$ id	E.type = lookup(id.entry)
$E \rightarrow E_1 \% E_2$	E.type = if E_1.type == integer and E_2.type==integer then integer else type_error
$E \rightarrow E_1[E_2]$	E.type = if E_2.type==integer and E_1.type==array(s,t) then t else type_error
$E \rightarrow *E_1$	E.type = if E_1.type==pointer(t) then t else type_error

Type Checking for Statement

$S \rightarrow id := E$

$S \rightarrow \text{if } E \text{ then } S1$

$S \rightarrow \text{while } E \text{ do } S1$

$S \rightarrow S1 ; S2$

Type Checking for statement

- Statements typically do not have values. Special basic type void can be assigned to them.

S → id := E	S.Type = if id.type == E.type then void else type_error
S → if E then S1	S.Type = if E.type == boolean then S1.type else type_error
S → while E do S1	S.Type = if E.type == boolean then S1.type else type_error
S → S1 ; S2	S.Type = if S1.type == void and S2.type == void then void else type_error