Lecture 17

Semantics Analysis

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

February 28, 2025

Take aways from the last class

- Bottom-up parsing of translation scheme

Take aways from the last class

- Bottom-up parsing of translation scheme
- Inherited Attributes on the parser stack

Take aways from the last class

- Bottom-up parsing of translation scheme
- Inherited Attributes on the parser stack
- Simulating the evaluation of inherited attributes

Take aways from the last class

- Bottom-up parsing of translation scheme
- Inherited Attributes on the parser stack
- Simulating the evaluation of inherited attributes
- General Algorithm

Type System

- A type is a set of values

Type System

- A type is a set of values
- Certain operations are legal for values of each type

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- A language's type system specifies which operations are valid for a type

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- 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

- A type is a set of values
- Certain operations are legal for values of each type
- 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
- Languages can be divided into three categories with respect to the type:

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- 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
- Languages can be divided into three categories with respect to the type:
  - Untyped: No type.

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- 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
- Languages can be divided into three categories with respect to the type:
  - Untyped: No type. Example Assembly

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- 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
- Languages can be divided into three categories with respect to the type:
  - Untyped: No type. Example Assembly
  - Statically typed: type assigned at compile time.

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- 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
- Languages can be divided into three categories with respect to the type:
  - Untyped: No type. Example Assembly
  - Statically typed: type assigned at compile time.Ex C,C++

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- 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
- Languages can be divided into three categories with respect to the type:
  - Untyped: No type. Example Assembly
  - Statically typed: type assigned at compile time.Ex C,C++
  - Dynamically typed: type checking at runtime.

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- 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
- Languages can be divided into three categories with respect to the type:
  - Untyped: No type. Example Assembly
  - Statically typed: type assigned at compile time.Ex C,C++
  - Dynamically typed: type checking at runtime.Ex Python

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- 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
- Languages can be divided into three categories with respect to the type:
  - Untyped: No type. Example Assembly
  - Statically typed: type assigned at compile time.Ex C,C++
  - Dynamically typed: type checking at runtime.Ex Python
- Static Typing

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- 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
- Languages can be divided into three categories with respect to the type:
  - Untyped: No type. Example Assembly
  - Statically typed: type assigned at compile time.Ex C,C++
  - Dynamically typed: type checking at runtime.Ex Python
- Static Typing
  - Catches most common programming errors at compile time

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- 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
- Languages can be divided into three categories with respect to the type:
  - Untyped: No type. Example Assembly
  - Statically typed: type assigned at compile time.Ex C,C++
  - Dynamically typed: type checking at runtime.Ex Python
- Static Typing
  - Catches most common programming errors at compile time
  - Avoids runtime overhead

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- 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
- Languages can be divided into three categories with respect to the type:
  - Untyped: No type. Example Assembly
  - Statically typed: type assigned at compile time.Ex C,C++
  - Dynamically typed: type checking at runtime.Ex Python
- Static Typing
  - Catches most common programming errors at compile time
  - Avoids runtime overhead
  - May be restrictive in some situations

Type System

- A type is a set of values
- Certain operations are legal for values of each type
- 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
- Languages can be divided into three categories with respect to the type:
  - Untyped: No type. Example Assembly
  - Statically typed: type assigned at compile time.Ex C,C++
  - Dynamically typed: type checking at runtime.Ex Python
- Static Typing
  - Catches most common programming errors at compile time
  - Avoids runtime overhead
  - May be restrictive in some situations
  - Rapid prototyping may be difficult

Rapid prototyping is an iterative approach to user interface design that can help you test and validate ideas early in the design process.

Type System and Type Checking

- A type system is a collection of rules for assigning type expressions to various parts of a program

Type System and Type Checking

- 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

Type System and Type Checking

- 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
- Types of `type`

Type System and Type Checking

- 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
- Types of `type`
  - Basic Type: integer, char, float, etc

Type System and Type Checking

- 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
- Types of `type`
  - Basic Type: integer, char, float, etc
  - Enumerated type: (voilet,indigo, red)

Type System and Type Checking

- 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
- Types of `type`
  - Basic Type: integer, char, float, etc
  - Enumerated type: (voilet,indigo, red)
  - Constructed Type: array, record, pointers, functions

Type Expression

- Type of a language construct is denoted by a type expression

Type Expression

- Type of a language construct is denoted by a type expression
- It is either a basic type
  or

Type Expression

- Type of a language construct is denoted by a type expression
- It is either a basic type
  or
  it is formed by applying operators called type constructor to other type expressions.

Type Expression

- Type of a language construct is denoted by a type expression
- It is either a basic type
  or
  it is formed by applying operators called type constructor to other type expressions.
- A type constructor applied to a type expression is a type expression

Type Expression

- Type of a language construct is denoted by a type expression
- It is either a basic type
  or
  it is formed by applying operators called type constructor to other type expressions.
- A type constructor applied to a type expression is a type expression
- A basic type is type expression. There are two other special basic types:

Type Expression

- Type of a language construct is denoted by a type expression
- It is either a basic type
  or
  it is formed by applying operators called type constructor to other type expressions.
- A type constructor applied to a type expression is a type expression
- A basic type is type expression. There are two other special basic types:
  - type error: error during type checking

Type Expression

- Type of a language construct is denoted by a type expression
- It is either a basic type
  or
  it is formed by applying operators called type constructor to other type expressions.
- A type constructor applied to a type expression is a type expression
- A basic type is type expression. There are two other special basic types:
  - type error: error during type checking
  - void: no type value

Type Constructor

- Array: 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

Type Constructor

- Array: 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
  var $A : array[1 \cdots 10]$ of integer

Type Constructor

- Array: 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
  $var \quad A : array[1 \cdots 10]$ of integer
  A has type expression $array(1 \cdots 10, integer)$

Type Constructor

- Array: 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
  $var \quad A : array[1 \cdots 10]$ of integer
  A has type expression $array(1 \cdots 10, integer)$
- Product: if $T1$ and $T2$ are type expressions, then their Cartesian product $T1xT2$ is a type expression

Type Constructor

- Array: 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
  $var \quad A : array[1 \cdots 10]$ of integer
  A has type expression $array(1 \cdots 10, integer)$
- Product: if $T1$ and $T2$ are type expressions, then their Cartesian product $T1xT2$ is a type expression
- Pointer: if T is a type expression, then pointer(T) is a type expression denoting type pointer to an object of type T

Type Constructor

- Array: 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
  $var \quad A : array[1 \cdots 10]$ of integer
  A has type expression $array(1 \cdots 10, integer)$
- Product: if $T1$ and $T2$ are type expressions, then their Cartesian product $T1xT2$ is a type expression
- Pointer: if T is a type expression, then pointer(T) is a type expression denoting type pointer to an object of type T
- Function: function maps domain set to range set. It is denoted by type expression $D \rightarrow R$

Type Constructor

- Array: 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
  $var \quad A : array[1 \cdots 10]$ of integer
  A has type expression $array(1 \cdots 10, integer)$
- Product: if $T1$ and $T2$ are type expressions, then their Cartesian product $T1xT2$ is a type expression
- Pointer: if T is a type expression, then pointer(T) is a type expression denoting type pointer to an object of type T
- Function: function maps domain set to range set. It is denoted by type expression $D \rightarrow R$
  $function \quad f(char\ a, char\ b) : *integer$; is denoted by $char \times char \rightarrow pointer(integer)$

Type Constructor

- Records: it applies to a tuple formed from field names and field types. Consider the declaration

Type Constructor

- Records: it applies to a tuple formed from field names and field types. Consider the declaration
  `type row = record`

Type Constructor

- Records: it applies to a tuple formed from field names and field types. Consider the declaration

```
type row = record
    addr :  integer;
```

Type Constructor

- Records: it applies to a tuple formed from field names and field types. Consider the declaration

```
type row = record
    addr :  integer;
    lexeme :  array [1 ..  15] of char
```

Type Constructor

- Records: it applies to a tuple formed from field names and field types. Consider the declaration

```
type row = record
    addr :  integer;
    lexeme :  array [1 ..  15] of char
end;
```

Type Constructor

- Records: it applies to a tuple formed from field names and field types. Consider the declaration

```
type row = record
    addr :  integer;
    lexeme :  array [1 ..  15] of char
end;
var table:  array [1 ..  10] of row;
```

Type Constructor

- Records: it applies to a tuple formed from field names and field types. Consider the declaration

```
type row = record
    addr :  integer;
    lexeme :  array [1 ..  15] of char
end;
var table:  array [1 ..  10] of row;
```

- type expression of table is array($1 \cdots 10$, row)

Type Constructor

- Records: it applies to a tuple formed from field names and field types. Consider the declaration

```
type row = record
    addr :  integer;
    lexeme :  array [1 ..  15] of char
end;
var table:  array [1 ..  10] of row;
```

- type expression of table is array($1 \cdots 10$, row)
- The type row has type expression

Type Constructor

- Records: it applies to a tuple formed from field names and field types. Consider the declaration

```
type row = record
    addr :  integer;
    lexeme :  array [1 ..  15] of char
end;
var table:  array [1 ..  10] of row;
```

- type expression of table is array($1 \cdots 10$, row)
- The type row has type expression
- record $((\text{addr} \times \text{integer}) \times (\text{lexeme} \times \textit{array}(1 \cdots 15, \text{char})))$

type expression for record.

Rules for Symbol Table entry

- $D \rightarrow id : T \quad addtype(id.entry, T.type)$

Rules for Symbol Table entry

- $D \rightarrow id : T \quad addtype(id.entry, T.type)$
  $T \rightarrow char \quad T.type = char$

Rules for Symbol Table entry

- $D \rightarrow id : T \quad addtype(id.entry, T.type)$
  $T \rightarrow char \quad T.type = char$
  $T \rightarrow integer \quad T.type = int$

Rules for Symbol Table entry

- $D \rightarrow id : T \quad addtype(id.entry, T.type)$
  $T \rightarrow char \quad T.type = char$
  $T \rightarrow integer \quad T.type = int$
  $T \rightarrow^{*} T_1 \quad T.type = pointer(T_1.type)$

Rules for Symbol Table entry

- $D \rightarrow id : T \quad addtype(id.entry, T.type)$
  $T \rightarrow char \quad T.type = char$
  $T \rightarrow integer \quad T.type = int$
  $T \rightarrow^{*} T_1 \quad T.type = pointer(T_1.type)$
  $T \rightarrow array[num] \text{ of } T_1 \quad T.type = array(1 \cdots num, T_1.type)$

Rules for Symbol Table entry

- $D \rightarrow id : T \quad addtype(id.entry, T.type)$
  $T \rightarrow char \quad T.type = char$
  $T \rightarrow integer \quad T.type = int$
  $T \rightarrow^{*} T_1 \quad T.type = pointer(T_1.type)$
  $T \rightarrow array[num] \text{ of } T_1 \quad T.type = array(1 \cdots num, T_1.type)$
- Type Checking for Function

Rules for Symbol Table entry

- $D \rightarrow id : T \quad addtype(id.entry, T.type)$
  $T \rightarrow char \quad T.type = char$
  $T \rightarrow integer \quad T.type = int$
  $T \rightarrow^* T_1 \quad T.type = pointer(T_1.type)$
  $T \rightarrow array[num] \text{ of } T_1 \quad T.type = array(1 \cdots num, T_1.type)$
- Type Checking for Function
  $E \rightarrow E1(E2)$

Rules for Symbol Table entry

- $D \rightarrow id : T \quad addtype(id.entry, T.type)$
  $T \rightarrow char \quad T.type = char$
  $T \rightarrow integer \quad T.type = int$
  $T \rightarrow^{*} T_1 \quad T.type = pointer(T_1.type)$
  $T \rightarrow array[num] \text{ of } T_1 \quad T.type = array(1 \cdots num, T_1.type)$
- Type Checking for Function
  $E \rightarrow E1(E2)$
  $E.type = if \ E2.type == s \ and \ E1.type == s \rightarrow t$

Rules for Symbol Table entry

- $D \rightarrow id : T \quad addtype(id.entry, T.type)$
  $T \rightarrow char \quad T.type = char$
  $T \rightarrow integer \quad T.type = int$
  $T \rightarrow^{*} T_1 \quad T.type = pointer(T_1.type)$
  $T \rightarrow array[num] \text{ of } T_1 \quad T.type = array(1 \cdots num, T_1.type)$
- Type Checking for Function
  $E \rightarrow E1(E2)$
  $E.type = if \ E2.type == s \ and \ E1.type == s \rightarrow t$
  $then \ t$

Rules for Symbol Table entry

- $D \rightarrow id : T \quad addtype(id.entry, T.type)$
  $T \rightarrow char \quad T.type = char$
  $T \rightarrow integer \quad T.type = int$
  $T \rightarrow^* T_1 \quad T.type = pointer(T_1.type)$
  $T \rightarrow array[num] \text{ of } T_1 \quad T.type = array(1 \cdots num, T_1.type)$
- Type Checking for Function
  $E \rightarrow E1(E2)$
  $E.type = if \ E2.type == s \ and \ E1.type == s \rightarrow t$
  $then \ t$
  $else \ type\_error$

Type checking for expressions

- $E \rightarrow literal \quad E.type = char$

Type checking for expressions

- $E \rightarrow literal \quad E.type = char$
- $E \rightarrow num \quad E.type = integer$

Type checking for expressions

- $E \rightarrow literal \quad E.type = char$
- $E \rightarrow num \quad E.type = integer$
- $E \rightarrow id \quad E.type = lookup(id.entry)$

Type checking for expressions

- $E \rightarrow literal \quad E.type = char$
- $E \rightarrow num \quad E.type = integer$
- $E \rightarrow id \quad E.type = lookup(id.entry)$
- $E \rightarrow E_1 \ mod \ E_2 \quad E.type = if \ E_1.type == integer \ and$
  $E_2.type == integer$
  $then \ integer$
  $else \ type\_error$

Type checking for expressions

- $E \rightarrow literal \quad E.type = char$
- $E \rightarrow num \quad E.type = integer$
- $E \rightarrow id \quad E.type = lookup(id.entry)$
- $E \rightarrow E_1 \ mod \ E_2 \quad E.type = if \ E_1.type == integer \ and$
  $E_2.type == integer$
  $then \ integer$
  $else \ type\_error$
- $E \rightarrow E_1[E_2] \quad E.type = if \ E_2.type == integer \ and$
  $E_1.type == array(s, t)$
  $then \ t$
  $else \ type\_error$

Type checking for expressions

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

E2 will denote the index of array.

Type checking for statements

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

Type checking for statements

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

- $S \rightarrow id = E \quad S.Type = if \ id.type == E.type$
  $then \quad void$
  $else \quad type\_error$

Type checking for statements

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

- $S \rightarrow id = E \quad S.Type = if \ id.type == E.type$
  $then \quad void$
  $else \quad type\_error$
- $S \rightarrow if(E) then \ S_1 \quad S.Type = if \ E.type == boolean$
  $then \ S_1.type$
  $else \ type\_error$

Type checking for statements

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

- $S \rightarrow id = E \quad S.Type = if \ id.type == E.type$
  $then \quad void$
  $else \quad type\_error$
- $S \rightarrow if(E) then \ S_1 \quad S.Type = if \ E.type == boolean$
  $then \ S_1.type$
  $else \ type\_error$
- $S \rightarrow while(E) do \ S_1 \quad S.Type = if \ E.type == boolean$
  $then \ S_1.type$
  $else \ type\_error$

Type checking for statements

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

- $S \rightarrow id = E \quad S.Type = if \ id.type == E.type$
  $then \quad void$
  $else \quad type\_error$
- $S \rightarrow if(E) then \ S_1 \quad S.Type = if \ E.type == boolean$
  $then \ S_1.type$
  $else \ type\_error$
- $S \rightarrow while(E) do \ S_1 \quad S.Type = if \ E.type == boolean$
  $then \ S_1.type$
  $else \ type\_error$
- $S \rightarrow S1; S2 \quad S.Type = if \ S1.type == void$
  $and \ S2.type == void$
  $then \ void$
  $else \ type\_error$