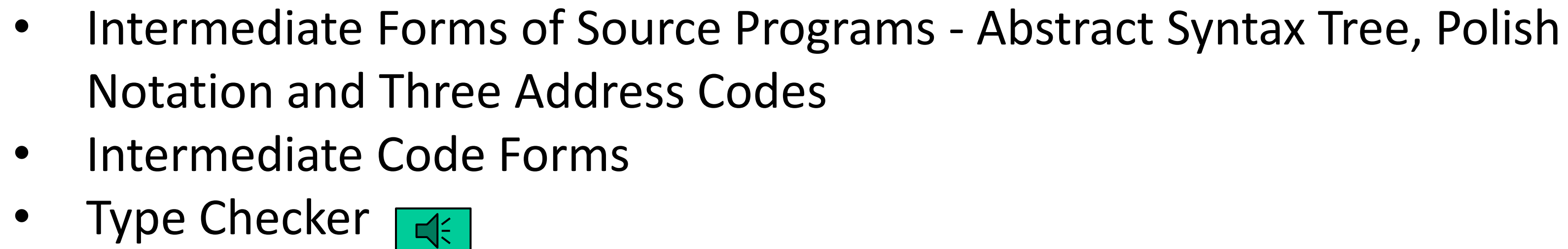
Unit - III

SEMANTIC ANALYSIS, INTERMEDIATE CODE GENERATOR & SYMBOL TABLE

SEMANTIC ANALYSIS

- Attributed Grammars
- Syntax Directed Translation

INTERMEDIATE CODE GENERATOR



- Intermediate Forms of Source Programs - Abstract Syntax Tree, Polish Notation and Three Address Codes
- Intermediate Code Forms
- Type Checker

SYMBOL TABLE

- Symbol Table Format
- Organization for Block Structures Languages
- Hashing

Dr. R. Madana Mohana

Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning

CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY

Hyderabad - 500 075, Telangana, INDIA

www.cbit.ac.in

Type Checking

Outline:

- Rules for Type Checking
- Type Conversions
- Overloading of Functions and Operators
- Type Inference and Polymorphic Functions

Type Checking

- ***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. For example, the **&&** operator in **Java** expects its two operands to be booleans; the result is also of type boolean.
- To do ***type checking*** a compiler needs to assign a type expression to each component of the source program.
- The compiler must then determine that these type expressions conform to a collection of logical rules that is called the type system for the source language.
- ***Type checking*** has the potential for catching errors in programs.

Type Checking

- In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of the element.
- A ***sound*** type system eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs.
- An implementation of a language is ***strongly typed*** if a compiler guarantees that the programs it accepts will run without type errors.

Type Checking

- Besides their use for compiling, ideas from ***type checking*** have been used to improve the security of systems that allow software modules to be imported and executed.
- **Java** programs compile into machine-independent bytecodes that include detailed type information about the operations in the bytecodes. Imported code is checked before it is allowed to execute, to guard against both inadvertent errors and malicious misbehavior.

Rules for Type Checking

Type checking can take on two forms: ***synthesis*** and ***inference***.

Type synthesis:

- ***Type synthesis*** builds up the type of an expression from the types of its subexpressions.
- It requires names to be declared before they are used.
- The type of $E_1 + E_2$ is defined in terms of the types of E_1 and E_2.

Rules for Type Checking

Type synthesis:

A typical rule for ***type synthesis*** has the form

> **if** f **has type** $s \rightarrow t$ **and** x **has type** s,
>
> **then expression** $f(x)$ **has type** t

Here, f and x denote expressions, and $s \rightarrow t$ denotes a function from s to t.

This rule for functions with one argument carries over to functions with several arguments.

This rule can be adapted for $E_1 + E_2$ by viewing it as a function application $add(E_1, E_2)$.

Rules for Type Checking

Type inference:

Type inference determines the type of a language construct from the way it is used.

Variables representing type expressions allow us to talk about unknown types.

We shall use Greek letters α, β, … for type variables in type expressions.

A typical rule for ***Type inference*** has the form

if $f(x)$ **is an expression,**

then for some α **and** β**,** f **has type** $\alpha \rightarrow \beta$ **and** x **has type** α

Rules for Type Checking

Type inference:

Type inference is needed for languages like **ML**, which check types, but do not require names to be declared.

In this section, we consider type checking of expressions. The rules for checking statements are similar to those for expressions.

For example, we treat the conditional statement "**if (*E*) *S*;**" as if it were the application of a function if to ***E*** and ***S***. Let the special type ***void*** denote the absence of a value. Then function ***if*** expects to be applied to a ***boolean*** and a ***void***; the result of the application is a ***void***.

Type Conversions

Consider expressions like ***x + i***, where ***x*** is of type float and ***i*** is of type integer. Since the representation of integers and floating-point numbers is different within a computer and different machine instructions are used for operations on integers and floats, the compiler may need to convert one of the operands of **+** to ensure that both operands are of the same type when the addition occurs.

Suppose that integers are converted to floats when necessary, using a unary operator (**float**).

For example, the integer **2** is converted to a float in the code for the expression **2 * 3.14**:

t_1 = (float) 2

t_2 = t_1 * 3.14

Type Conversions

We can extend such examples to consider integer and float versions of the operators; for example, **int*** for integer operands and **float*** for floats.

Type synthesis will be illustrated by extending the scheme discussed above for translating expressions. We introduce another attribute **E.type**, whose value is either ***integer*** or ***float***. The rule associated with $E \rightarrow E_1 + E_2$ builds on the pseudocode:

if (E_1.type = *integer* **and** E_2.type = *integer*) E.type = *integer*;

else if (E_1.type = *float* **and** E_2.type = *integer*) …

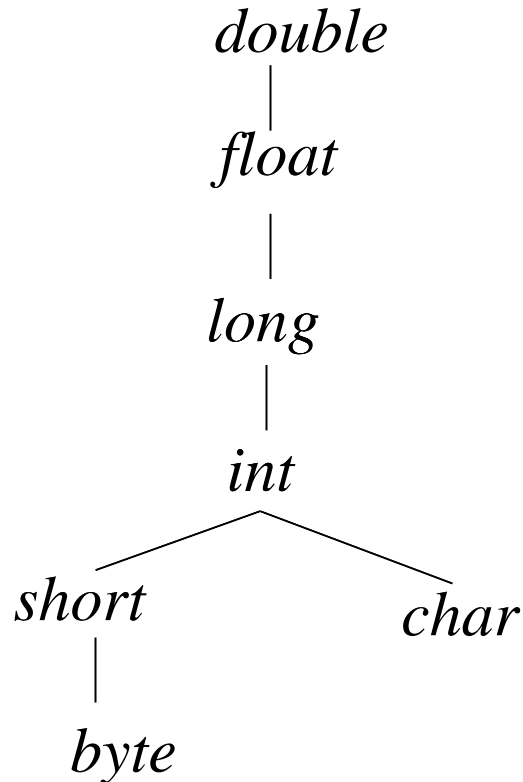
Type Conversions

As the number of types subject to conversion increases, the number of cases increases rapidly. Therefore with large numbers of types, careful organization of the semantic actions becomes important.

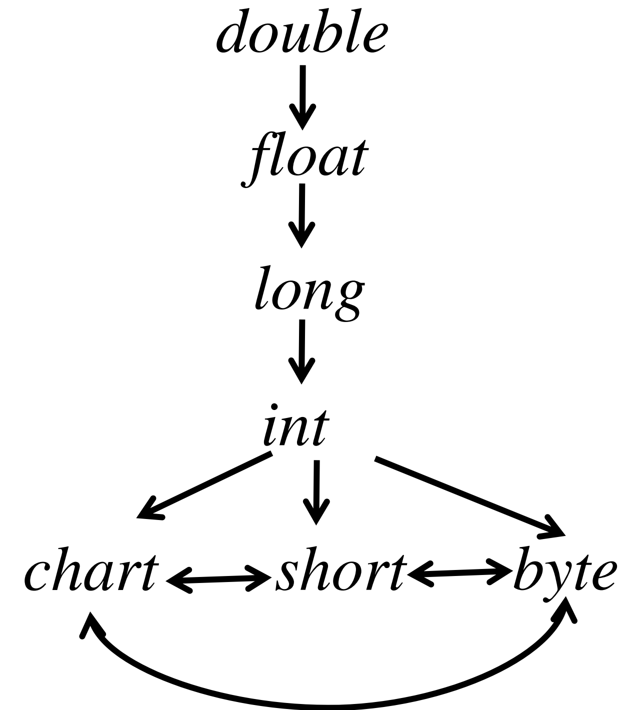
Type conversion rules vary from language to language.

The rules for Java shown in below Fig. distinguish between widening conversions, which are intended to preserve information, and narrowing conversions, which can lose information.

Type Conversions



(a) Widening conversions



(b) Narrowing conversions

Fig. Conversions between primitive types in Java

Type Conversions

- Conversion from one type to another is said to be ***implicit*** if it is done automatically by the compiler.
- Implicit type conversions, also called ***coercions***, are limited in many languages to widening conversions.
- Conversion is said to be ***explicit*** if the programmer must write something to cause the conversion.
- Explicit conversions are also called ***casts.***

Type Conversions

The semantic action for checking $E \rightarrow E_1 + E_2$ uses <u>two functions</u>:

1. $max(t_1, t_2)$ takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy. It declares an error if either t_1 or t_2 is not in the hierarchy; e.g., if either type is an array or a pointer type.

Type Conversions

2. *widen(a, t, w)* generates type conversions if needed to widen the contents of an address a of type ***t*** into a value of type ***w***. It returns a itself if ***t*** and ***w*** are the same type. Otherwise, it generates an instruction to do the conversion and place the result in a temporary, which is returned as the result. Pseudocode for ***widen***, assuming that the only types are ***integer*** and ***float***, shown below:

```
Addr widen(Addr a, Type t, Type w)
        if (t = w) return a;
        else if (t = integer and w = float){
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

Type Conversions

Introducing type conversions into expression evaluation:

$E \rightarrow E_1 + E_2$	{ E.type = max(E₁.type; E₂.type);

Overloading of Functions and Operators

- An **overloaded** symbol has different meanings depending on its context.
- Overloading is **resolved** when a unique meaning is determined for each occurrence of a name.
- In this section, we restrict attention to overloading that can be resolved by looking only at the arguments of a function, as in **Java**.

Overloading of Functions and Operators

Example:

The **+** operator in **Java** denotes either string concatenation or addition, depending on the types of its operands. User-defined functions can be overloaded as well, as in

```
void err() {…}
void err(String s) {…}
```

Note that we can choose between these two versions of a function **err** by looking at their arguments.

Overloading of Functions and Operators

Example:

The following is a **type-synthesis** rule for overloaded functions:

if f can have type $s_i \rightarrow t_i$, for $1 \leq i \leq n$, where $s_i \neq s_j$ for $i \neq j$

and x has type s_k, for some $1 \leq k \leq n$

then expression $f(x)$ has type t_k

Type Inference and Polymorphic Functions

- ***Type inference*** is useful for a language like ***Machine Language (ML)***, which is strongly typed, but does not require names to be declared before they are used.
- ***Type inference*** ensures that names are used consistently.
- The term "**polymorphic**" refers to any code fragment that can be executed with arguments of different types.
- In this section, we consider parametric polymorphism, where the polymorphism is characterized by parameters or type variables.

Type Inference and Polymorphic Functions

- The running example is the ***ML*** program shown in *Fig.* , which defines a function ***length***.
- The type of ***length*** can be described as, "for any type, length maps a list of elements of type to an integer."

```
fun length(x)=
    if null(x)then 0
    else length(tl(x))+ 1; // tl-tail
```

Fig. **ML** program for the length of a list

Type Inference and Polymorphic Functions

Example-1:

- In *above Fig.* , the keyword **`fun`** introduces a function definition; functions can be recursive.
- The program fragment defines function length with one parameter **`x`**.
- The body of the function consists of a conditional expression.
- The predefined function **`null`** tests whether a list is empty, and the predefined function **`tl`** (short for ”tail”) returns the remainder of a list after the first element is removed.

Type Inference and Polymorphic Functions

Example-1:

- The function ***length*** determines the length or number of elements of a list ***x***.
- All elements of a list must have the same type, but ***length*** can be applied to lists whose elements are of any one type.
- In the following expression, ***length*** is applied to two different types of lists (list elements are enclosed within "**[**" and "**]**"):

```
length(["sun", "mon", "tue"]) + length([10, 9, 8, 7])
```

The list of strings has length **3** and the list of integers has length **4**, so the above expression evaluates to **7**.

Type Inference and Polymorphic Functions

Example-2:

The following example informally infers a type for ***length***, implicitly using type inference rules, which is repeated here:

if ***f(x)*** **is an expression,**

then for some α **and** β, ***f*** **has type** $\alpha \rightarrow \beta$ **and** ***x*** **has type** α

Type Inference and Polymorphic Functions

Example-2:

Abstract syntax tree for the function definition shown below:

```
fun length(x)=
    if null(x) then 0
    else length(tl(x))+ 1; // tl-tail
```

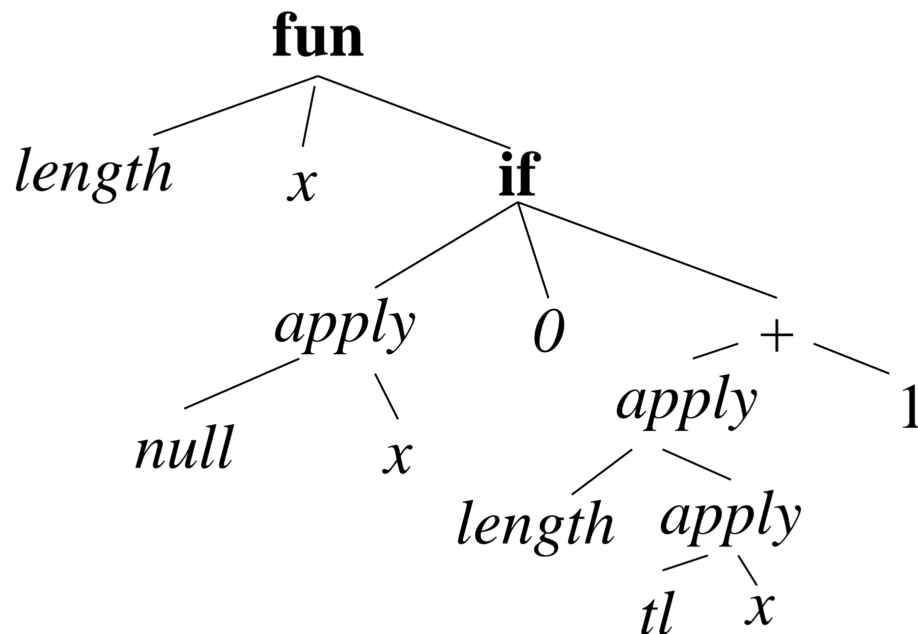


Fig. Abstract syntax tree for the above function definition

Type Inference and Polymorphic Functions

Substitutions, Instances, and Unification:

If ***t*** is a type expression and ***S*** is a ***substitution*** (a mapping from type variables to type expressions), then we write ***S(t)*** for the result of consistently replacing all occurrences of each type variable **α** in ***t*** by **S(α)**. **S(t)** is called an ***instance*** of ***t***.

For example, ***list(integer)*** is an instance of ***list(α)***, since it is the result of substituting ***integer*** for **α** ***list(α)***.

Note, however, that ***integer → float*** is not an instance of **α → α**, since a substitution must replace all occurrences of **α** by the same type expression.

Type Inference and Polymorphic Functions

Substitutions, Instances, and Unification:

Substitution S is a ***unifier*** of type expressions t_1 and t_2 if $S(t_1) = S(t_2)$. S is the ***most general unifier*** of t_1 and t_2 if for any other unifier of t_1 and t_2, say S', it is the case that for any t, $S'(t)$ is an instance of $S(t)$. In words, S' imposes more constraints on t than S does.

Type Inference and Polymorphic Functions

Algorithm: Type inference for polymorphic functions

INPUT: A program consisting of a sequence of function definitions followed by an expression to be evaluated. An expression is made up of function applications and names, where names can have predefined polymorphic types.

OUTPUT: Inferred types for the names in the program.

METHOD: For simplicity, we shall deal with unary functions only. The type of a function $f(x_1, x_2)$ with two parameters can be represented by a type expression $s_1 \times s_2 \rightarrow t$, where s_1 and s_2 are the types of x_1 and x_2, respectively, and t is the type of the result $f(x_1, x_2)$. An expression $f(a, b)$ can be checked by matching the type of a with s_1 and the type of b with s_2.

Type Inference and Polymorphic Functions

Check the function definitions and the expression in the input sequence. Use the inferred type of a function if it is subsequently used in an expression.

- For a function definition **fun id_1(id_2) = *E***, create fresh type variables α and β. Associate the type $\alpha \rightarrow \beta$ with the function **id_1**, and the type α with the parameter **id_2**. Then, infer a type for expression ***E***. Suppose α denotes type ***s*** and β denotes type ***t*** after type inference for ***E***. The inferred type of function **id_1** is $s \rightarrow t$. Bind any type variables that remain unconstrained in $s \rightarrow t$ by $\forall$ quantifiers.

Type Inference and Polymorphic Functions

- For a function application $E_1(E_2)$, infer types for E_1 and E_2. Since E_1 is used as a function, its type must have the form $s \rightarrow s'$. (Technically, the type of E_1 must unify with $\beta \rightarrow \gamma$, where β and γ are new type variables). Let t be the inferred type of E_2. Unify s and t. If unification fails, the expression has a type error. Otherwise, the inferred type of $E_1(E_2)$ is s'.
- For each occurrence of a polymorphic function, replace the bound variables in its type by distinct fresh variables and remove the $\forall$ quantifiers. The resulting type expression is the inferred type of this occurrence.
- For a name that is encountered for the first time, introduce a fresh variable for its type.

Type Inference and Polymorphic Functions

Inferring a type for the function ***length*** shown below:

```
fun length(x)=
    if null(x) then 0
    else length(tl(x))+ 1; // tl-tail
```

Type Inference and Polymorphic Functions

Inferring a type for the function ***length*** shown below:

LINE	EXPRESSION : TYPE	UNIFY
1)	*length* : $\beta \rightarrow \gamma$	
2)	*x* : β	
3)	**if** : *boolean* X α_i X $\alpha_i \rightarrow \alpha_i$	
4)	*null* : *list*(α_n) $\rightarrow$ *boolean*	
5)	*null(x)* : *boolean*	*list*(α_n) = β
6)	*0* : *integer*	α_i = *integer*
7)	**+** : *integer* X *integer* $\rightarrow$ *integer*	
8)	*tl* : *list*(α_t) $\rightarrow$ *list*(α_t)	
9)	*tl(x)* : *list*(α_t)	*list*(α_t) = *list*(α_n)
10)	*length(tl(x))* : γ	γ = *integer*
11)	*1* : *integer*	
12)	*length(tl(x)) + 1* : *integer*	
13)	*if(...)* : *integer*	

Summary

<u>Type Checking</u>

- Rules for Type Checking
- Type Conversions
- Overloading of Functions and Operators
- Type Inference and Polymorphic Functions

Reading: Aho2, Section 6.5.1 to 6.5.5

Next Lecture: SYMBOL TABLE