Module 4

Syntax Directed Translation: Syntax Directed definitions, Bottom up Evaluation of S attributed definitions, L attributed definitions, Top Down translation, Bottom up evaluation of Inherited attributes

Type Checking: Type Systems, Specification of a simple type checker

Need for Semantic Analysis

Semantic analysis is a pass by a compiler that adds semantic information to the parse tree and performs certain checks based on this information. It logically follows the parsing phase, in which the parse tree is generated, and logically precedes the code generation phase, in which (intermediate/target) code is generated. (In a compiler implementation, it may be possible to fold different phases into one pass.) Typical examples of semantic information that is added and checked is typing information (type checking) and the binding of variables and function names to their definitions (object binding). Sometimes also some early code optimization is done in this phase. For this phase the compiler usually maintains *symbol tables* in which it stores what each symbol (variable names, function names, etc.) refers to.

Following things are done in Semantic Analysis:

1.**Disambiguate Overloaded operators** : If an operator is overloaded, one would like to specify the meaning of that particular operator because from one will go into code generation phase next.

2. **Type checking** : The process of verifying and enforcing the constraints of types is called type checking. This may occur either at compile-time (a static check) or run-time (a dynamic check). Static type checking is a primary task of the semantic analysis carried out by a compiler. If type rules are enforced strongly (that is, generally allowing only those automatic type conversions which do not lose information), the process is called strongly typed, if not, weakly typed.

3. **Uniqueness checking** : Whether a variable name is unique or not, in the its scope.

4. **Type coercion** : If some kind of mixing of types is allowed. Done in languages which are not strongly typed. This can be done dynamically as well as statically.

5. **Name Checks** : Check whether any variable has a name which is not allowed. Ex. Name is same as an identifier(Ex. int in java).

A parser has its own limitations in catching program errors related to semantics, something that is deeper than syntax analysis. Typical features of semantic analysis cannot be modeled using context free grammar formalism. If one tries to incorporate those features in the definition of a language then that language doesn't remain context free anymore**.** These are a couple of examples which tell us that typically what a compiler has to do beyond syntax analysis. An identifier **x** can be declared in two separate functions in the program, once of the type int and then of the type char. Hence the same identifier will have to be bound to these two different properties in the two different contexts.

What does a compiler need to know during semantic analysis?

Whether a variable has been declared?

Are there variables which have not been declared?

What is the type of the variable?

Whether a variable is a scalar, an array, or a function?

What declaration of the variable does each reference use?

If an expression is type consistent?

If an array use like A[i,j,k] is consistent with the declaration? Does it have three dimensions?

For example, we have the third question from the above list, i.e., what is the type of a variable and we have a statement like

int a, b , c;

Then we see that syntax analyzer cannot alone handle this situation. We actually need to traverse the parse trees to find out the type of identifier and this is all done in semantic analysis phase. Purpose of listing out the questions is that unless we have answers to these questions we will not be able to write a semantic analyzer. This becomes a feedback mechanism.

If the compiler has the answers to all these questions only then will it be able to successfully do a semantic analysis by using the generated parse tree. These questions give a feedback to what is to be done in the semantic analysis. These questions help in outlining the work of the semantic analyzer. In order to answer the previous questions the compiler will have to keep information about the type of variables, number of parameters in a particular function etc. It will have to do some sort of computation in order to gain this information. Most compilers keep a structure called symbol table to store this information. At times the information required is not available locally, but in a different scope altogether. In syntax analysis we used context free grammar. Here we put lot of attributes around it. So it consists of context sensitive grammars along with extended attribute grammars. Ad-hoc methods also good as there is no structure in it and the formal method is simply just too tough. So we would like to use something in between. Formalism may be so difficult that writing specifications itself may become tougher than writing compiler itself. So we do use attributes but we do analysis along with parse tree itself instead of using context sensitive grammars.

Syntax Directed Translation

The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree. By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.

– We associate Attributes to the grammar symbols representing the language constructs.

– Values for attributes are computed by Semantic Rules associated with grammar productions.

Evaluation of Semantic Rules may:

– Generate Code;

– Insert information into the Symbol Table;

– Perform Semantic Check;

– Issue error messages;

– etc.

There are two notations for attaching semantic rules:

1. **Syntax Directed Definitions**: High-level specification hiding many implementation details (also called Attribute Grammars).

2. **Translation Schemes:** More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

Syntax Directed Definitions

Syntax Directed Definitions are a generalization of context-free grammars in which:

1. Grammar symbols have an associated set of Attributes;

2. Productions are associated with Semantic Rules for computing the values of attributes.

• Such formalism geneates Annotated Parse-Trees where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X).

The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

• We distinguish between two kinds of attributes:

1. **Synthesized Attributes:** They are computed from the values of the attributes of the children nodes.

2. **Inherited Attributes**: They are computed from the values of the attributes of both the siblings and the parent nodes.

Form of Syntax Directed Definitions

• Each production, $A \rightarrow \alpha$, is associated with a set of semantic rules: b := f(c1, c2, . . . , ck), where f is a function and either

1. b is a synthesized attribute of A, and c1, c2, . . . , ck are attributes of the grammar symbols of the production, or

2. b is an inherited attribute of a grammar symbol in α, and c1, c2, . . . , ck are attributes of grammar symbols in α or attributes of A.

• Note. Terminal symbols are assumed to have synthesized attributes supplied by the lexical analyzer.

• Procedure calls (e.g. print in the next slide) define values of Dummy synthesized attributes of the non terminal on the left-hand side of the production.

Example: Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called val.

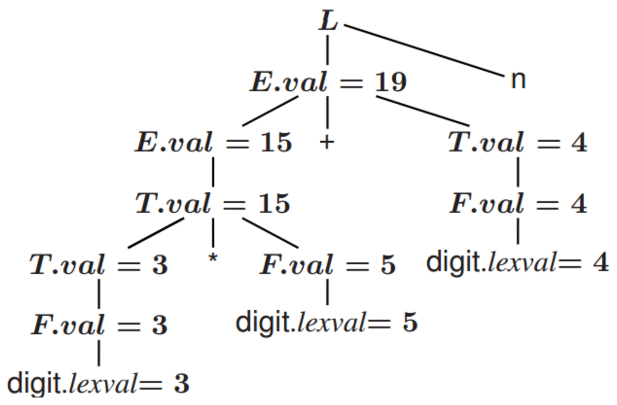
PRODUCTION	SEMANTIC RULE
$L \to E\text{n}$	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \text{digit}$	$F.val := \text{digit}.lexval$

S-Attributed Definitions

Definition. An S-Attributed Definition is a Syntax Directed Definition that uses only synthesized attributes.

• Evaluation Order : Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.

• Example. The above arithmetic grammar is an example of an S-Attributed Definition. The annotated parse-tree for the input 3*5+4n is:



Inherited Attributes

• Inherited Attributes are useful for expressing the dependence of a construct on the context in which it appears.

• It is always possible to rewrite a syntax directed definition to use only synthesized attributes, but it is often more natural to use both synthesized and inherited attributes.

• Evaluation Order. Inherited attributes cannot be evaluated by a simple PreOrder traversal of the parse-tree:

• Unlike synthesized attributes, the order in which the inherited attributes of the children are computed is important. Indeed: Inherited attributes of the children can depend from both left and right siblings.

Implementing Syntax Directed Definitions
– Dependency Graphs
– S-Attributed Definitions
– L-Attributed Definitions

Dependency Graphs

• Implementing a Syntax Directed Definition consists primarily in finding an order for the evaluation of attributes

– Each attribute value must be available when a computation is performed.

• Dependency Graphs are the most general technique used to evaluate syntax directed definitions with both synthesized and inherited attributes.

• A Dependency Graph shows the interdependencies among the attributes of the various nodes of a parse-tree.

– There is a node for each attribute;

– If attribute b depends on an attribute c there is a link from the node for c to the node for b (b ← c).

• Dependency Rule: If an attribute b depends from an attribute c, then we need to fire the semantic rule for c first and then the semantic rule for b.

Evaluation Order

• The evaluation order of semantic rules depends from a Topological Sort derived from the dependency graph.

• Topological Sort: Any ordering m1, m2, . . . , mk such that if mi → mj is a link in the dependency graph then mi < mj .

• Any topological sort of a dependency graph gives a valid order to evaluate the semantic rules.

Evaluation of S-Attributed Definitions

• Synthesized Attributes can be evaluated by a bottom-up parser as the input is being analyzed avoiding the construction of a dependency graph.

• The parser keeps the values of the synthesized attributes in its stack.

• Whenever a reduction $A \rightarrow \alpha$ is made, the attribute for A is computed from the attributes of α which appear on the stack.

• Thus, a translator for an S-Attributed Definition can be simply implemented by extending the stack of an LR-Parser.

•Extra fields are added to the stack to hold the values of synthesized attributes.

• In the simple case of just one attribute per grammar symbol the stack has two
fields: state and val

state	*val*
Z	$Z.x$
Y	$Y.x$
X	$X.x$
...	...

• The current top of the stack is indicated by the pointer top.

• Synthesized attributes are computed just before each reduction:

– Before the reduction $A \rightarrow XY\ Z$ is made, the attribute for A is computed:

A.a := f(val[top], val[top − 1], val[top − 2]).

L-Attributed Definitions

• L-Attributed Definitions contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them.

• Definition. A syntax directed definition is L-Attributed if each inherited attribute of Xj in a production $A \rightarrow X1 \ldots Xj \ldots Xn$, depends only on:

1. The attributes of the symbols to the left (this is what L in L-Attributed stands for) of Xj , i.e., $X1X2 \ldots Xj{-}1$, and

2. The inherited attributes of A.

• Theorem. Inherited attributes in L-Attributed Definitions can be computed by a PreOrder traversal of the parse-tree.

Evaluating L-Attributed Definitions

L-Attributed Definitions are a class of syntax directed definitions whose attributes can always be evaluated by single traversal of the parse-tree.

• The following procedure evaluate L-Attributed Definitions by mixing PostOrder (synthesized) and PreOrder (inherited) traversal.

```
Algorithm: L-Eval(n: Node)
Input: Node of an annotated parse-tree.
Output: Attribute evaluation.

Begin
    For each child m of n, from left-to-right Do
    Begin
        Evaluate inherited attributes of m;
        L-Eval(m)
    End;
    Evaluate synthesized attributes of n
End.
```

Translation Schemes

• Translation Schemes are more implementation oriented than syntax directed definitions since they indicate the order in which semantic rules and attributes are to be evaluated.

• Definition. A Translation Scheme is a context-free grammar in which

1. Attributes are associated with grammar symbols;
2. Semantic Actions are enclosed between braces {} and are inserted within the right-hand side of productions.

• Yacc uses Translation Schemes.

• Translation Schemes deal with both synthesized and inherited attributes.

• Semantic Actions are treated as terminal symbols: Annotated parse-trees contain semantic actions as children of the node standing for the corresponding production.

• Translation Schemes are useful to evaluate L-Attributed definitions at parsing time (even if they are a general mechanism).

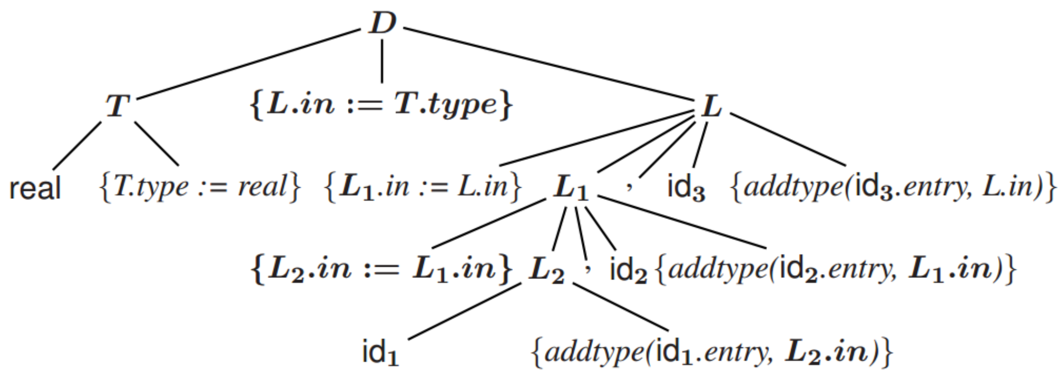
– An L-Attributed Syntax-Directed Definition can be turned into a Translation Scheme.

Translation Schemes: An Example

• Consider the Translation Scheme for the L-Attributed Definition for “type declarations”:

$$
\begin{array}{ll}
D \to & T\ \{L.in := T.type\}\ L \\
T \to & \mathsf{int}\ \ \{T.type := \mathit{integer}\} \\
T \to & \mathsf{real}\ \ \{T.type := \mathit{real}\} \\
L \to & \{L_1.in := L.in\}\ L_1, \mathsf{id}\ \ \{\mathit{addtype}(\mathsf{id}.\mathit{entry}, L.in)\} \\
L \to & \mathsf{id}\ \ \{\mathit{addtype}(\mathsf{id}.\mathit{entry}, L.in)\}
\end{array}
$$

The parse-tree with semantic actions for the input real id1, id2, id3 is:



Traversing the Parse-Tree in depth-first order (PostOrder) we can evaluate the attributes.

Type Checking

• A compiler must check that the program follows the Type Rules of the language.

• Information about Data Types is maintained and computed by the compiler.

• The Type Checker is a module of a compiler devoted to type checking tasks.

• Examples of Tasks:

1. The operator mod is defined only if the operands are integers;

2. Indexing is allowed only on an array and the index must be an integer;

3. A function must have a precise number of arguments and the parameters must have a correct type;

4. etc...

• Type Checking may be either static or dynamic: The one done at compile time is static.

• In languages like Pascal and C type checking is primarily static and is used to check the correctness of a program before its execution.

• Static type checking is also useful to determine the amount of memory needed to store variables.
• The design of a Type Checker depends on the syntactic structure of language constructs, the Type Expressions of the language, and the rules for assigning types to constructs.

Type Expressions

• A Type Expression denotes the type of a language construct.
• A type expression is either a Basic Type or is built applying Type Constructors to other types.
1. A Basic Type is a type expression (int, real, boolean, char). The basic type void represents the empty set and allows statements to be checked;
2. Type expressions can be associated with a name: Type Names are type expressions;
3. A Type Constructor applied to type expressions is a type expression.

Type constructors inlude:
(a) Array. If T is a type expression, then array(I, T) is a type expression denoting an array with elements of type T and index range in I—e.g., array[1..10] of int == array(1..10,int);
(b) Product. If T1 e T2 are type expressions, then their Cartesian Product T1 × T2 is a type expression;
(c) Record. Similar to Product but with names for different fields (used to access the components of a record). Example of a C record type:

```
struct
{
        double r;
        int i;
}
```

(d) Pointer. If T is a type expression, then pointer(T) is the type expression "pointer to an object of type T ";
(e) Function. If D is the domain and R the range of the function then we denote its type by the type expression: D : R.
The mod operator has type, int × int : int.

Type System

• Type System: Collection of rules for assigning type expressions to the various part of a program.

• Type Systems are specified using syntax directed definitions.

• A type checker implements a type system.

• Definition. A language is strongly typed if its compiler can guarantee that the program it accepts will execute without type errors.

Specification of a Type Checker

• We specify a type checker for a simple language where identifiers have an associated type.

• Attribute Grammar for Declarations and Expressions:

$$P \to D; E$$

$$D \to D; D \mid \textsf{id} : T$$

$$T \to \textsf{char} \mid \textsf{int} \mid \textsf{array}[\textsf{num}] \textsf{ of } T \mid \uparrow T$$

$$E \to \textsf{literal} \mid \textsf{num} \mid \textsf{id} \mid E \textsf{ mod } E \mid E[E] \mid E\uparrow$$

The syntax directed definition for associating a type to an Identifier is:

PRODUCTION	SEMANTIC RULE
$D \to \textsf{id} : T$	$addtype(\textsf{id}.entry, T.type)$
$T \to \textsf{char}$	$T.type := char$
$T \to \textsf{int}$	$T.type := int$
$T \to \uparrow T_1$	$T.type := pointer(T_1.type)$
$T \to \textsf{array}[\textsf{num}] \textsf{ of } T_1$	$T.type := array(1..\textbf{num}.val, T_1.type)$

• All the attributes are synthesized.

• Since P → D; E, all the identifiers will have their types saved in the symbol table before type checking an expression E.

The syntax directed definition for associating a type to an Expression is:

PRODUCTION	SEMANTIC RULE
$E \to$ **literal**	$E.type := char$
$E \to$ **num**	$E.type := int$
$E \to$ **id**	$E.type := lookup(\textbf{id}.entry)$
$E \to E_1$ **mod** E_2	$E.type :=$ if $E_1.type = int$ **and** $E_2.type = int$ then int else $type\_error$
$E \to E_1[E_2]$	$E.type :=$ if $E_2.type = int$ **and** $E_1.type = array(i,t)$ then t else $type\_error$
$E \to E_1\uparrow$	$E.type :=$ if $E_1.type = pointer(t)$ then t else $type\_error$

The syntax directed definition for associating a type to a Statement is:

PRODUCTION	SEMANTIC RULE
$S \to$ **id** $:= E$	$S.type :=$ if id.$type$ = $E.type$ then $void$ else $type\_error$
$S \to$ if E then S_1	$S.type :=$ if $E.type$ = $boolean$ then $S_1.type$ else $type\_error$
$S \to$ while E do S_1	$S.type :=$ if $E.type$ = $boolean$ then $S_1.type$ else $type\_error$
$S \to S_1; S_2$	$S.type :=$ if $S_1.type = void$ **and** $S_2.type = void$ then $void$ else $type\_error$

• The type expression for a statement is either void or type error.

• For languages with type names or (even worst) allowing sub-typing we need to define when two types are equivalent.

The type checker for a function is:

PRODUCTION	SEMANTIC RULE
$D \rightarrow \mathbf{id} : T$	$addtype(\mathrm{id}.entry, T.type);\ D.type := T.type$
$D \rightarrow D_1 ; D_2$	$D.type := D_1.type \times D_2.type$
$Fun \rightarrow \mathbf{fun}\ \mathbf{id}(D) : T ; B$	$addtype(\mathrm{id}.entry, D.type : T.type)$
$B \rightarrow \{S\}$	
$S \rightarrow \mathbf{id}(EList)$	$E.type :=$ if $lookup(\mathrm{id}.entry) = t_1 : t_2$ and $EList.type = t_1$ then t_2 else $type\_error$
$EList \rightarrow E$	$EList.type := E.type$
$EList \rightarrow EList, E$	$EList.type := EList_1.type \times E.type$

Type Conversion

• Example. What's the type of "x + y" if:

1. x is of type real;
2. y is of type int;
3. Different machine instructions are used for operations on reals and integers.

• Depending on the language, specific conversion rules must be adopted by the compiler to convert the type of one of the operand of +.

– The type checker in a compiler can insert these conversion operators into the intermediate code.

– Such an implicit type conversion is called Coercion.

Type Coercion in Expressions

• The syntax directed definition for coercion from integer to real for a generic arithmetic operation op is:

PRODUCTION	SEMANTIC RULE
$E \rightarrow \mathbf{num}$	$E.type := int$
$E \rightarrow \mathbf{num.num}$	$E.type := real$
$E \rightarrow \mathbf{id}$	$E.type := lookup(\mathbf{id}.entry)$
$E \rightarrow E_1\ \mathbf{op}\ E_2$	$E.type :=$ if $E_1.type = int$ and $E_2.type = int$ then int else if $E_1.type = int$ and $E_2.type = real$ then $real$ else if $E_1.type = real$ and $E_2.type = int$ then $real$ else if $E_1.type = real$ and $E_2.type = real$ then $real$ else $type\_error$