chapter6: Introduction to Semantic Analysis; Syntax-Directed Translation

1. Introduction

- **Semantic Analysis Phase**
  - **Purpose**: compute additional information needed for compilation that is beyond the capabilities of Context-Free Grammars and Standard Parsing Algorithms
  - **It performs Static semantic analysis**: which takes place prior to execution
  - **The primarily tasks are:**

    Building a symbol table, performing type inference and type checking.
- The static semantic analysis is described by
  - **Attribute grammar**
    - identify **attributes** of language entities that must be computed
    - and to write **attribute equations** or **semantic rules** that express how the computation of such attributes is related to the grammar rules of the language
  - Attribute grammar is useful and applicable for languages that obey the principle of **Syntax-Directed Semantics.**
  - **Abstract syntax** as represented by an abstract syntax tree
- **Implementation** of the static semantic analysis:
  - Not as clearly expressible as parsing algorithms because of the addition problem caused by the timing of the analysis during the compilation process
  - Multi-pass (more common) or single pass lead to totally different process

2. Introduction to Type Checking

There is a level of correctness that goes beyond grammar and syntax.
example

```
extern int foo(int a,int b,int c,int d);

int fee()
{
int f[3],g[1],h,i,j,k;
char *p;
foo(h,i,"ab",j,k);
k = f*i+j;
h = g[17];
printf("%s,%s\n",p,q);
p = &(i+j);
k = 3.14159;
p = p*2;
}
```

Find the errors in the program.

- declared g[1], used g[17]
- wrong number of args to foo
- "ab" is not an int
- f declared as an array, used as a scalar
- 3.14159 is not an int
- & cannot be applied to a non-atomic expression
- * cannot be applied to pointers

None of these are simply syntax errors. They require a deeper analysis of the meaning (semantics) of the program.

3. Syntax Directed Definitions

The theoretical framework for semantic analysis uses *syntax directed definitions*, sometimes called *attribute grammars*. A syntax directed definition is an extension of a context free grammar in which

Each grammar symbol is assigned certain attributes
Each production is augmented with semantic rules which are used to define the values of attributes

The attributes give meaning to the structures in the parse tree, such as:

- the data type of an expression or variable
- the value of a constant expression
- the number and types of parameters expected by a function
- the memory location and size of an expression
- the symbol table associated with a compound statement
- code generated to execute a statement or expression

The process of computing the attributes of each node is called *annotating* the parse tree, which is then called an *annotated* or *attributed* parse tree (or syntax tree).
One problem for compiler writers is that language references do not normally provide an attribute grammar, only the pure syntax of the context-free grammar. It is left to the compiler writer to construct an attribute grammar from the English language descriptions of the components of the programming language.

4. Attributes and Attribute Grammars

- **Attributes**
  - Any property of a programming language construct such as
    - The data type of a variable
    - The value of an expression
    - The location of a variable in memory
    - The object code of a procedure
    - The number of significant digits in a number
- **Binding of the attribute**
  - The process of computing an attribute and associating its computed value with the language construct in question
- **Binding time**
  - The time during the compilation/execution process when the binding of an attribute occurs
  - Based on the difference of the binding time, attributes is divided into **Static** attributes (be bound prior to execution) and **Dynamic** attributes (be bound during execution)
- **Example:** The binding time and significance during compilation of the attributes.
  - Type checker
    - In a language like C or Pascal, is an important part of semantic analysis;
    - While in a language like LISP , data types are dynamic, LISP compiler must generate code to compute types and perform type checking during program execution.
  - The values of expressions
    - Usually dynamic and the be computed during execution;

- But sometime can also be evaluated during compilation (constant folding).

– The allocation of a variable:
  - Either static (such as in FORTRAN77) or dynamic (such as in LISP),
  - Sometimes it is a mixture of static and dynamic (such as in C and Pascal) depending on the language and properties of the variable itself.
– Object code of a procedure:
  - A static attribute, which is computed by the code generator
– Number of significant digits in a number:
  - Often not explicitly treated during compilation.

4.1 **Attribute Grammars**

- X.a means the value of 'a' associated to 'X'
  – X is a grammar symbol and a is an attribute associated to X
- **Syntax-directed semantics**:
  – Attributes are associated directly with the grammar symbols of the language.
  – Given a collection of attributes a1, …, ak, it implies that for each grammar rule X0→X1X2…Xn (X0 is a nonterminal),
  – The values of the attributes Xi.aj of each grammar symbol Xi are related to the values of the attributes of the other symbols in the rule.

- An **attribute grammar** for attributes a1, a2, … , ak is the collection of all **attribute equations** or **semantic rules** of the following form,
  – for all the grammar rules of the language.
    - Xi.aj = fij(X0.a1,…,X0.ak, …, X1.al, …, Xn-1.a1, …Xn.ak)
    - Where fij is a mathematical function of its arguments

- Typically, attribute grammars are written in tabular form as follows:

Grammar Rule	Semantic Rules
Rule 1	Associated attribute equations
...	
Rule n	Associated attribute equation

Example 6.1 consider the following simple grammar for unsigned numbers:

Number → number digit | digit
Digit → 0|1|2|3|4|5|6|7|8|9

The most significant attribute: numeric value (write as val), and the responding attribute grammar is as follows:

Grammar Rule	Semantic Rules
Number1→number2 digit	number1.val = number2.val*10+digit.val
Number→digit	number.val= digit.val
digit→0	digit.val = 0
digit→1	digit.val = 1
digit→2	digit.val = 2
digit→3	digit.val = 3
digit→4	digit.val = 4
digit→5	digit.val = 5
digit→6	digit.val = 6
digit→7	digit.val = 7
digit→8	digit.val = 8
digit→9	digit.val = 9

table 6.1

The parse tree showing attribute computations for the number 345 is given as follows

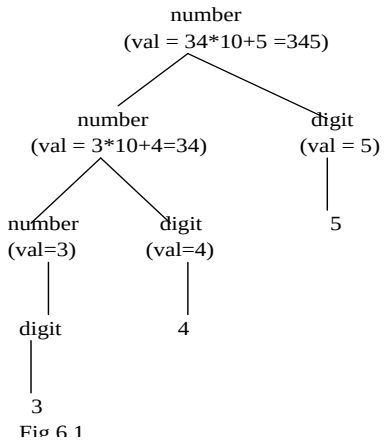


Fig 6.1

Example 6.2 consider the following grammar for simple integer arithmetic expressions:

Exp → exp + term | exp-term | term

Term → term*factor | factor

Factor → (exp)| number

The principal attribute of an exp (or term or factor) is its numeric value (write as val) and the attribute equations for the val attribute are given as follows

Grammar Rule	Semantic Rules
exp1→exp2+term	exp1.val=exp2.val+term.val
exp1→exp2-term	exp1.val=exp2.val-erm.val
exp1→ term	exp1.val= term.val
term1→term2*factor	term1.val=term2.val*factor.val
term→factor	term.val=factor.val
factor→(exp)	factor.val=exp.val
factor→number	factor.val=number.val

table 6.2

Given the expression (34-3)*42 , the computations implied by this attribute grammar by **attaching equations to nodes** in a parse tree is as follows

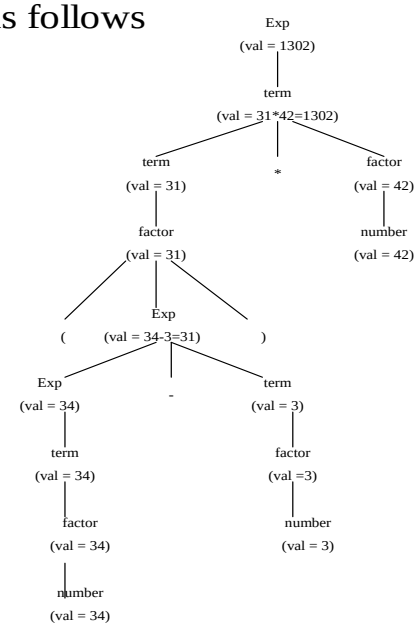


Fig6.2

Example 6.3 consider the following simple grammar of variable declarations in a C-like syntax:

Decl → type var-list
Type→int | float
Var-list→id, var-list |id

Define a data type attribute for the variables given by the identifiers in a declaration and write equations expressing how the data type attribute is related to the type of the declaration as follows:

(We use the name *dtype* to distinguish the attribute from the nonterminal type)

Grammar Rule	Semantic Rules
decl→type var-list	var-list.dtype = type.dtype
type→int	type.dtype = integer
type→ float	type.dtype = real
var-list1→id,var-list2	id.dtype = var-list1.dtype var-list2.dtype= var-list1.dtype
var-list→id	id.type = var-list.dtype

table 6.3

Note that **there is no equation involving the dtype of the nonterminal decl**.

It is not necessary for the value of an attribute to be specified for all grammar symbols

Parse tree for the string **float x,y** showing the dtype attribute as specified by the attribute grammar above is as follows:

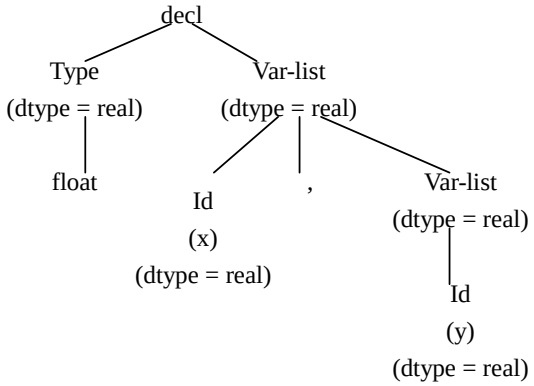


Fig6.3

Attribute grammars may involve several interdependent attributes.

Example 6.4 consider the following grammar, where numbers may be octal or decimal, suppose this is indicated by a one-character suffix o(for octal) or d(for decimal):

Based-num → num basechar
Basechar → o|d
Num → num digit | digit
Digit → 0|1|2|3|4|5|6|7|8|9

In this case **num and digit require a new attribute base**, which is used to compute the val attribute. The attribute grammar for base and val is given as follows.

Table 6.4 Attribute grammar for Example 6.4

Grammar Rule	Semantic Rules
based-num → *num basechar*	*based-num.val* = *num.val* *num.base* = *basechar.base*
basechar → **o**	*basechar.base* = 8
basechar → **d**	*basechar.base* = 10
$num_1 \rightarrow num_2\ digit$	$num_1.val$ = **if** *digit.val* = *error* **or** $num_2.val$ = *error* **then** *error* **else** $num_2.val * num_1.base + digit.val$ $num_2.base = num_1.base$ $digit.base = num_1.base$
num → *digit*	*num.val* = *digit.val* *digit.base* = *num.base*
digit → **0**	*digit.val* = 0
digit → **1**	*digit.val* = 1
. . .	. . .
digit → **7**	*digit.val* = 7
digit → **8**	*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 8
digit → **9**	*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 9

Two new features should be noted in this attribute grammar. First, the BNF grammar does not itself eliminate the erroneous combination of the (non-octal) digits **8** and **9** with the **o** suffix. For instance, the string **189o** is syntactically correct according to the above BNF, but cannot be assigned any value. Thus, a new *error* value is needed for such cases. Additionally, the attribute grammar must express the fact that the inclusion of **8** or **9** in a number with an **o** suffix results in an *error* value. The easiest way to do this is to use an **if-then-else** expression in the functions of the appropriate attribute equations. For instance, the equation

$$num_1.val = \textbf{if } digit.val = error \textbf{ or } num_2.val = error \textbf{ then } error \textbf{ else } num_2.val * num_1.base + digit.val$$

corresponding to the grammar rule $num_1 \rightarrow num_2\ digit$ expresses the fact that if either of $num_2.val$ or $digit.val$ are *error* then $num_1.val$ must also be *error*, and only if that is not the case is $num_1.val$ given by the formula $num_2.val * num_1.base + digit.val$.

To conclude this example, we again show the attribute calculations on a parse tree. Figure 6.4 gives a parse tree for the number **345o**, together with the attribute values computed according to the attribute grammar of Table 6.4.

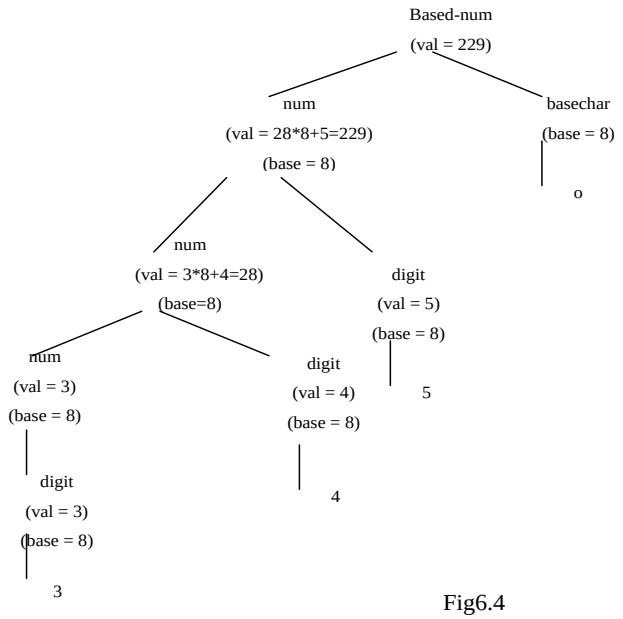


Fig6.4

4.2 Simplifications and Extensions to Attribute Grammars

- **Meta-language** for the attribute grammar:
  - The collection of expressions allowable in an attribute equation,
    - Arithmetic, logical and a few other kinds of expressions,
    - Together with an if-then-else expression and occasionally a case or switch expression
- **Functions** can be added to the meta-language whose definitions may be given elsewhere
  - For instance : digit → D (D is understood to be one of the digits) digit.val = numval(D)
    - here, numval is a function which is defined as :

- Int numval(char D)
- {return (int)D – (int)'0';}

Simplifications :

Using abstract syntax tree instead of parse tree,
(34-3)*42

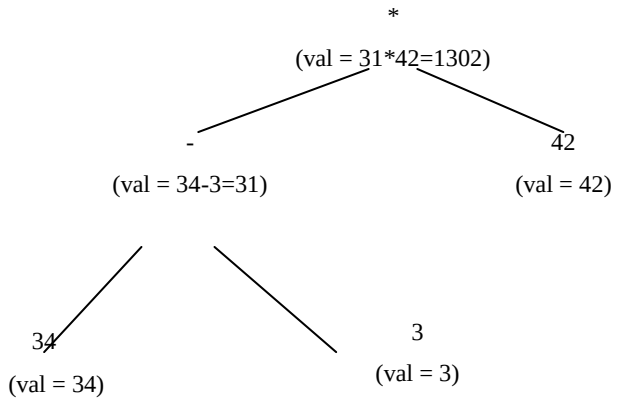


Fig 6.5

5. **Algorithms for Attribute Computation**

- **Purpose:** study the ways an attribute grammar can be used as basis for a compiler to compute and use the attributes defined by the equations of the attribute grammar.
  - Attribute equations is turned into computation rules
  - Xi.aj = fij(X0.a1,…,X0.ak, …, X1.al, …, Xn-1.a1, …Xn.ak)
  - is viewed as an assignment of the value of the functional expression on the right- hand side to the attribute Xi.aj.
- The attribute equations indicate the **order constraints on the computation** of the attributes.
  - Attribute at the right-hand side must be computed before that at the left-hand side.
  - The constraints is represented by directed graphs — **dependency graphs.**

5.1 **Dependency graphs and evaluation order**

- **Dependency graph** of the string:

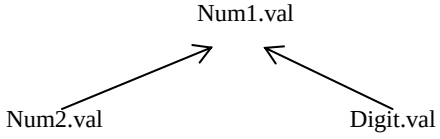
The union of the dependency graphs of the grammar rule choices representing each node (nonleaf) of the parse tree of the string

  - Xi.aj = fij(…, Xm.ak, …)
  - An edge from each node Xm.ak to Xi.aj the node expressing the dependency of Xi.aj on Xm.ak.

Example 6.6 Consider the grammar of Example 6.1, with the attribute grammar as given in tab 6.1. for each symbol there is only one node in each dependency graph, corresponding to its val attribute

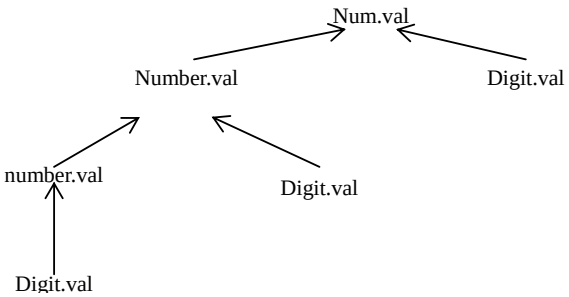
Grammar rule	attribute equation
Number1→number2 digit	number1.val = number2.val * 10 +digit.val

The dependency graph for this grammar rule choice is

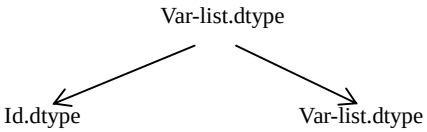


The subscripts for repeated symbols will be omitted
Number→ digit number.val = digit.val

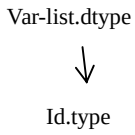
The string 345 has the following dependency graph.



Example 6.7 consider the grammar of example 6.3
var-list1 →id, varlist2
id.dtype = var-list1.dtype
var-list2.dtype = var-list1.dtype
and the dependency graph



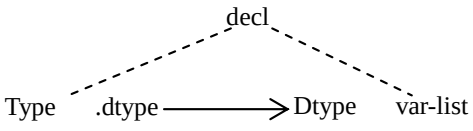
similarly var-list →id respond to



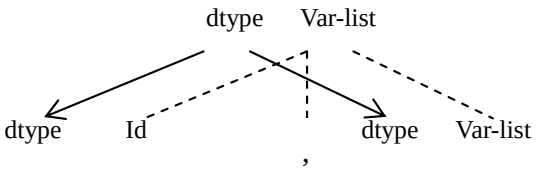
Decl→type varlist

Type.dtype ——> var-list.dtype

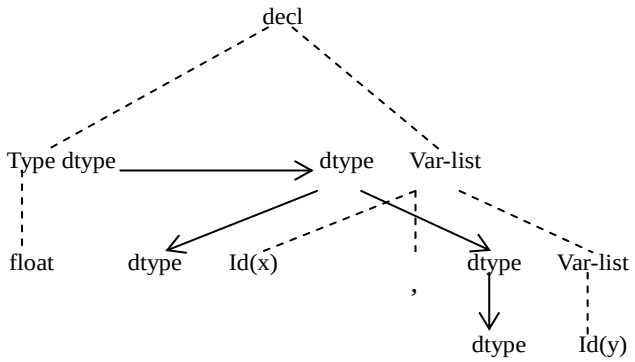
It can also be drawn as:



So, the first graph in this example can be drawn as :



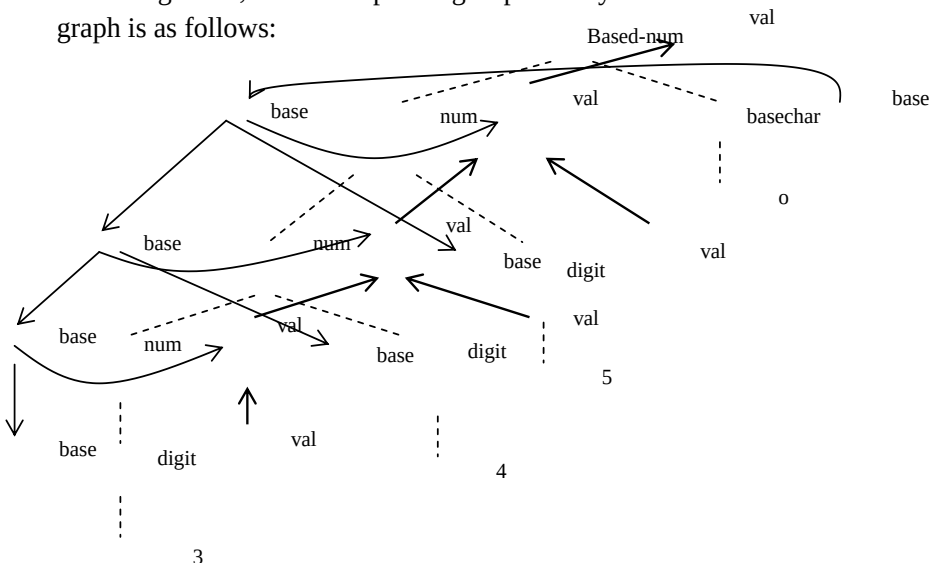
Finally, the dependency graph for the string float x, y is



Example 6.8 consider the grammar of based number of example 6.4

based-num→num basechar
Num→num digit
Num→digit
Digit→9
…

For string 345o, the corresponding dependency graph is as follows:



- Any algorithm must compute the attribute at each node in the dependency graph before it attempts to compute any successor attributes
  - A traversal order of the dependency graph that obeys this restriction is called a **topological sort**
  - Requiring that the graph must be **acyclic**, such graphs are called directed acyclic graphs Or **DAG**s

Example 6.9

The dependency of Fig 6.6 is a DAG, which gives as follows:

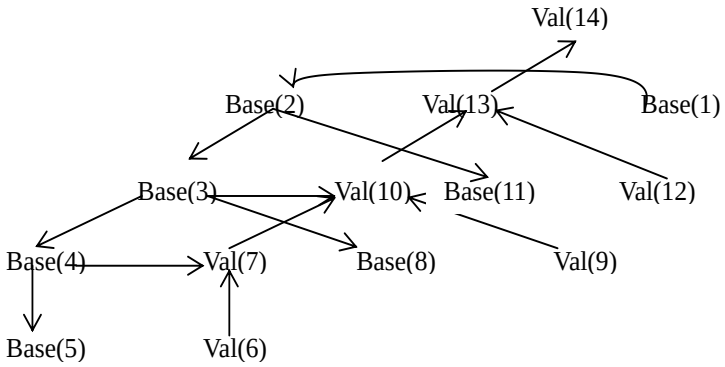


Fig 6.7

Another topological sort is given by the order

12 6 9 1 2 11 3 8 4 5 7 10 13 14

- One question: how attribute values are found at the roots of the graph (root of the graph mean that has no predecessors)
- Attribute values at these nodes is often in the form of tokens, and should be computed before any other attribute values
- **Parse tree method**:
  - Construction of the dependency graph is based on the specific parse tree at compile time, add complexity, and need circularity detective
- **Rule based method**:
  - Fix an order for attribute evaluation at compiler construction time

5.2 **Synthesized and Inherited Attributes**

- **Classification** of the attributes:

1. Synthesized attributes

- **Definition:**
  - An attribute is **synthesized** if all its dependencies point form child to parent in the parse tree.
  - Equivalently, an attribute a is synthesized if, given a grammar rule A→X1X2… Xn, the only associated attribute equation with an '**a**' on the left-hand side is of the form:
    - A.a = f(X1.a1,…X1.ak,…Xn.a1,… Xn.ak)

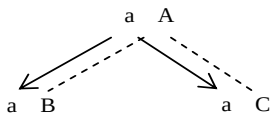
- **An S-attributed grammar**
  - An attribute grammar in which all the attributes are synthesized

2. Inherited attributes

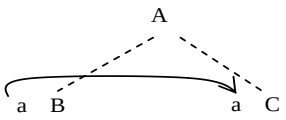
Definition:
An attribute that is not synthesized is called an **inherited** attribute.
Such as dtype in the example below:

Decl → type var-list
Type→int|float
Var-list→id,var-list|id

Two basic kinds of dependency of inherited attributes:

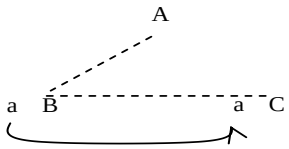


(a) Inheritance form parent to siblings



(b) Inheritance from sibling to sibling

If some structures in a syntax tree are implemented via sibling pointers, then sibling inheritance can proceed directly along a sibling chain, as follows:



(c) Sibling inheritance via sibling pointers

inherited attributes can be computed by a preorder traversal , or combined preorder/inorder traversal of the parse or syntax tree, represented by the following pseudocode:

```
Procedure PreEval(T: treenode);
Begin
    For each child C of T do
        Compute all inherited attributes of    T;
        PreEval(C);
End;
```