Compiler Principle and Technology

Prof. Dongming LU
Apr. 15th, 2015

6. Semantic Analysis

- **Semantic Analysis Phase**
  - **Purpose**: compute additional information needed for compilation that is beyond the capabilities of Context-Free Grammars and Standard Parsing Algorithms
  - **Static semantic analysis**: Take place prior to execution (Such as building a symbol table、performing type inference and type checking)

- **Classification**
  - Semantic Check: type check, array bound check …
  - Semantic processing: memory assignment, value computation …

- **Description** of the static semantic analysis
  - **Attribute grammar**
    - Identify **attributes** of language entities that must be computed
    - Write **attribute equations** or **semantic rules** that express how the computation of such attributes is related to the grammar rules of the language
  - The most useful 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

- **Emphasis:**
  - Attributes and attribute grammars
  - Algorithms for attribute computation
  - The symbol table
  - Data types and type checking

Contents

6.1 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.
  - Attribute computations are extremely varied
  - Type checker
    - In a language like C or Pascal, is an important part of semantic analysis;
    - 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 be computed during execution;
    - Sometime can also be evaluated during compilation (constant folding).

- **Example:** the binding time and significance during compilation of the attributes.
  - Attribute computations are extremely varied
  - 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.

6.1.1 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 $a_1, \ldots, a_k$, it implies that for each grammar rule $X_0 \rightarrow X_1X_2\ldots X_n$ (X_0 is a nonterminal), the values of the attributes $X_i.a_j$ of each grammar symbol X_i are related to the values of the attributes of the other symbols in the rule.

- An **attribute grammar** for attributes $a_1, \ldots, a_k$, is the collection of all **attribute equations** or **semantic rules** of the following form :
  - for all the grammar rules of the language.
    - $X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, \ldots, X_1.a_l, \ldots, X_{n-1}.a_1, \ldots X_n.a_k)$
    - Where f_{ij} 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

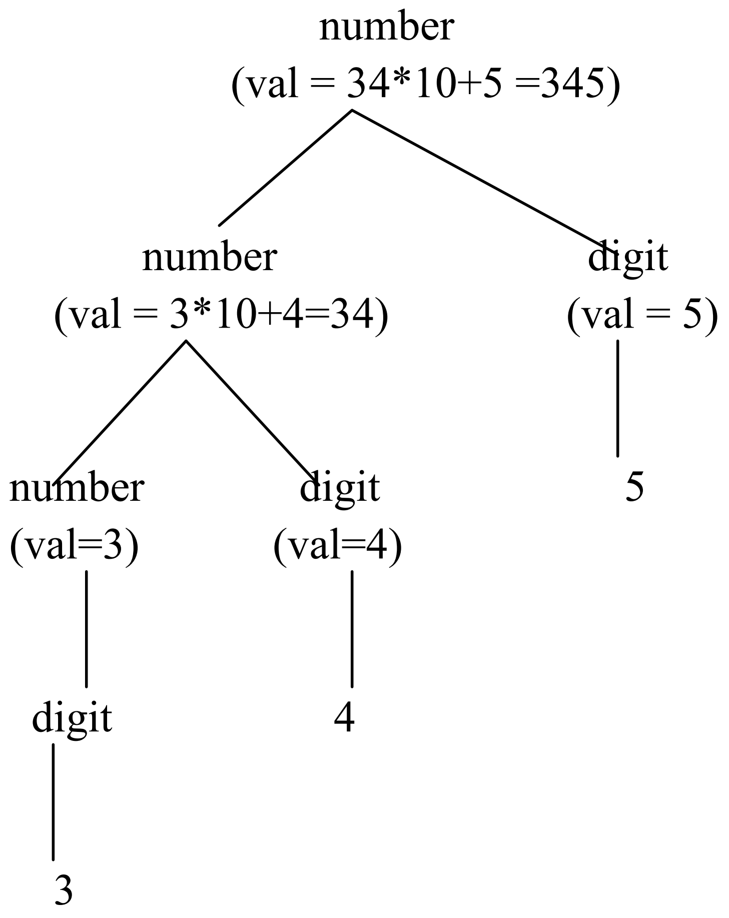

Fig 6.1

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

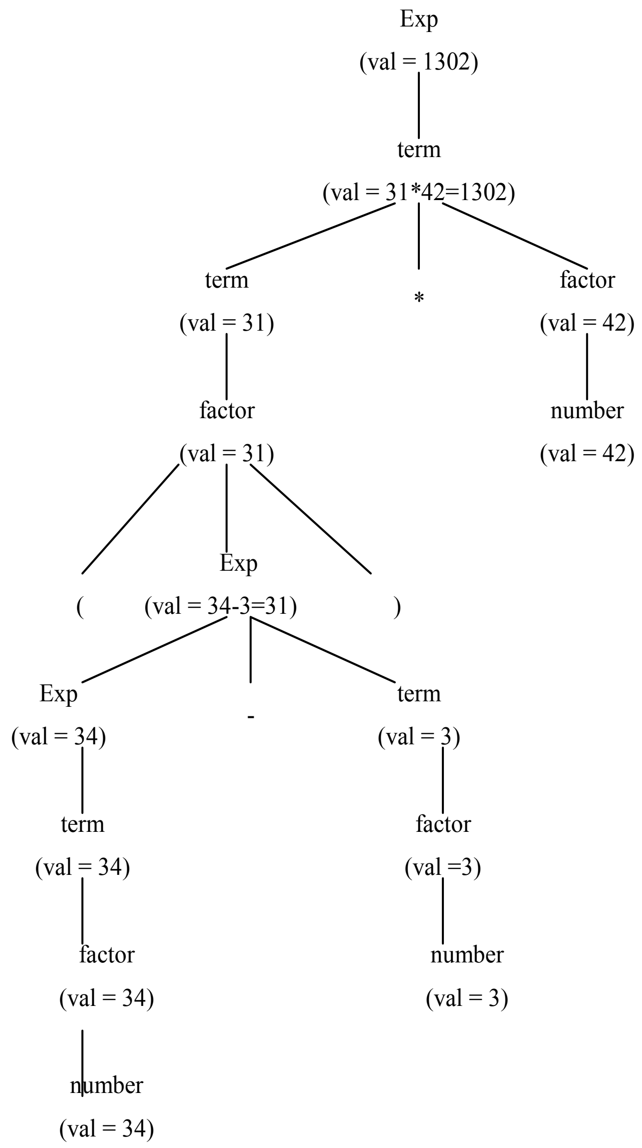

Fig6.2

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

- Write equations expressing how the data type attribute is related to the type of the declaration as follows:

(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: 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:

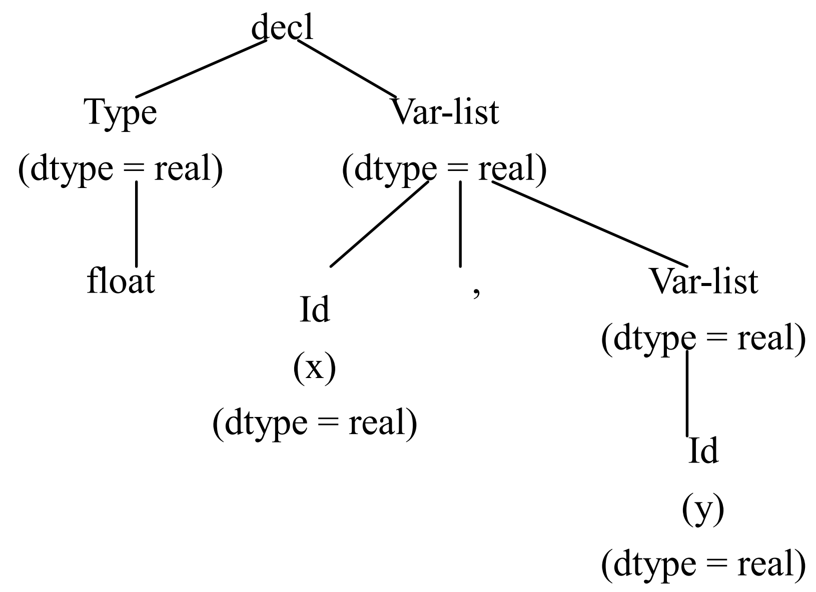

Fig6.3

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.

Grammar Rule	Semantic Rules
Based-num→num basechar	Based-num.val = num.val
	Based-num.base = basechar.base
Basechar →**o**	Basechar.base = 8
Basechar→ **d**	Basechar.base = 10
Num1→num2 digit	num1.val =
	if digit.val = error or num2.val = error
	Then error
	Else num2.val*num1.base+digit.val
	Num2.base = num1.base
	Digit.base = num1.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

Tab 6.4

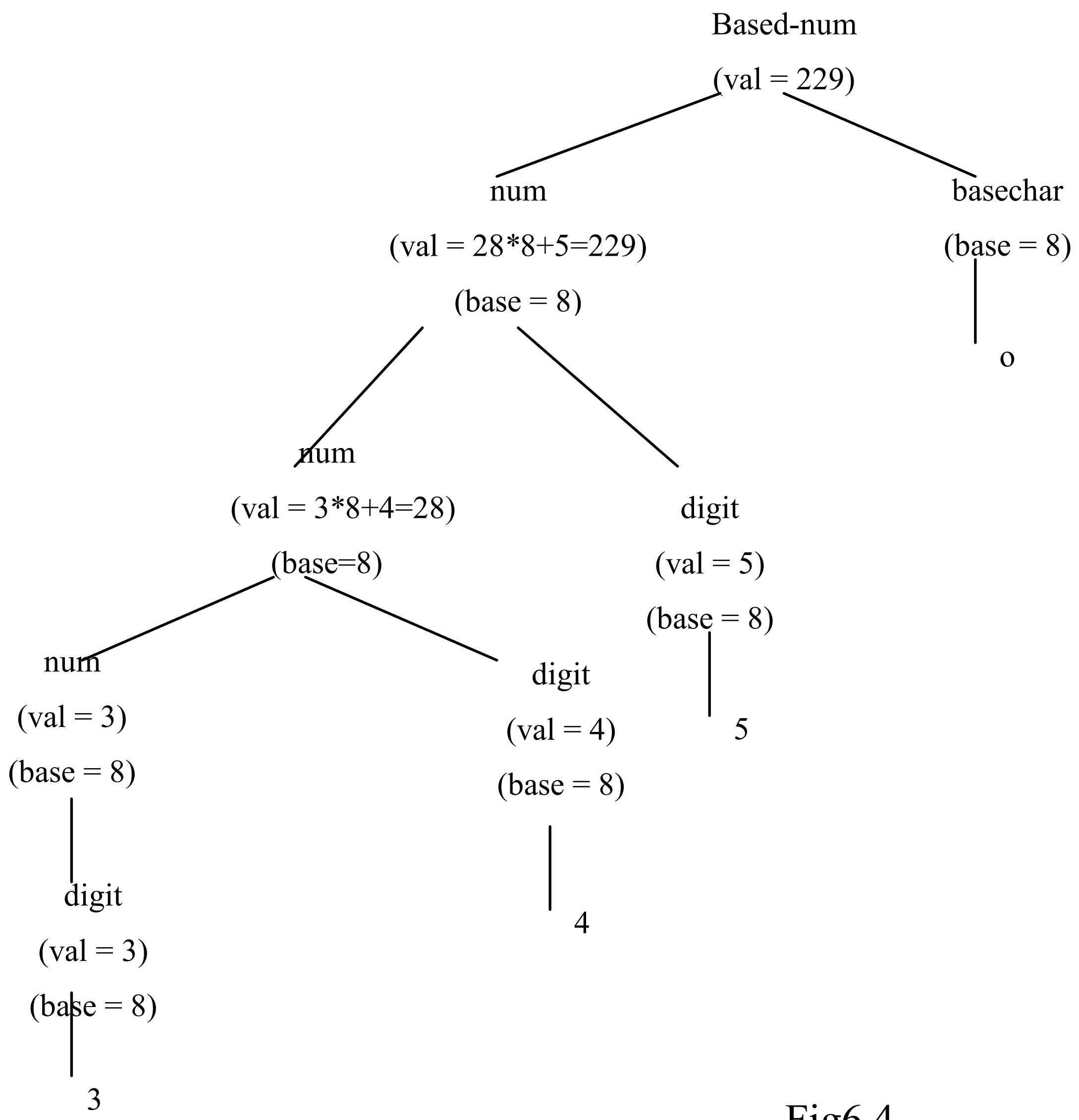
Based-num
(val = 229)
num
(val = 28*8+5=229)
(base = 8)
basechar
(base = 8)
o
num
(val = 3*8+4=28)
(base=8)
digit
(val = 5)
(base = 8)
5
num
(val = 3)
(base = 8)
digit
(val = 4)
(base = 8)
4
digit
(val = 3)
(base = 8)
3

Fig6.4

6.1.2 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 ambiguous grammar:

(all ambiguity will have been dealt with at the parser stage)

exp → exp + exp| exp – exp | exp * exp|(exp)|number

Grammar Rule	Semantic Rules
exp1→exp2+exp3	exp1.val=exp2.val+exp3.val
exp1→exp2-exp3	exp1.val=exp2.val-exp3.val
exp1→exp2*exp3	exp1.val=exp2.val*exp3.val
exp1→(exp2)	exp1.val=exp2.val
exp→number	exp.val=number.val

table 6.5

Simplifications :

Using abstract syntax tree instead of parse tree

(34-3)*42

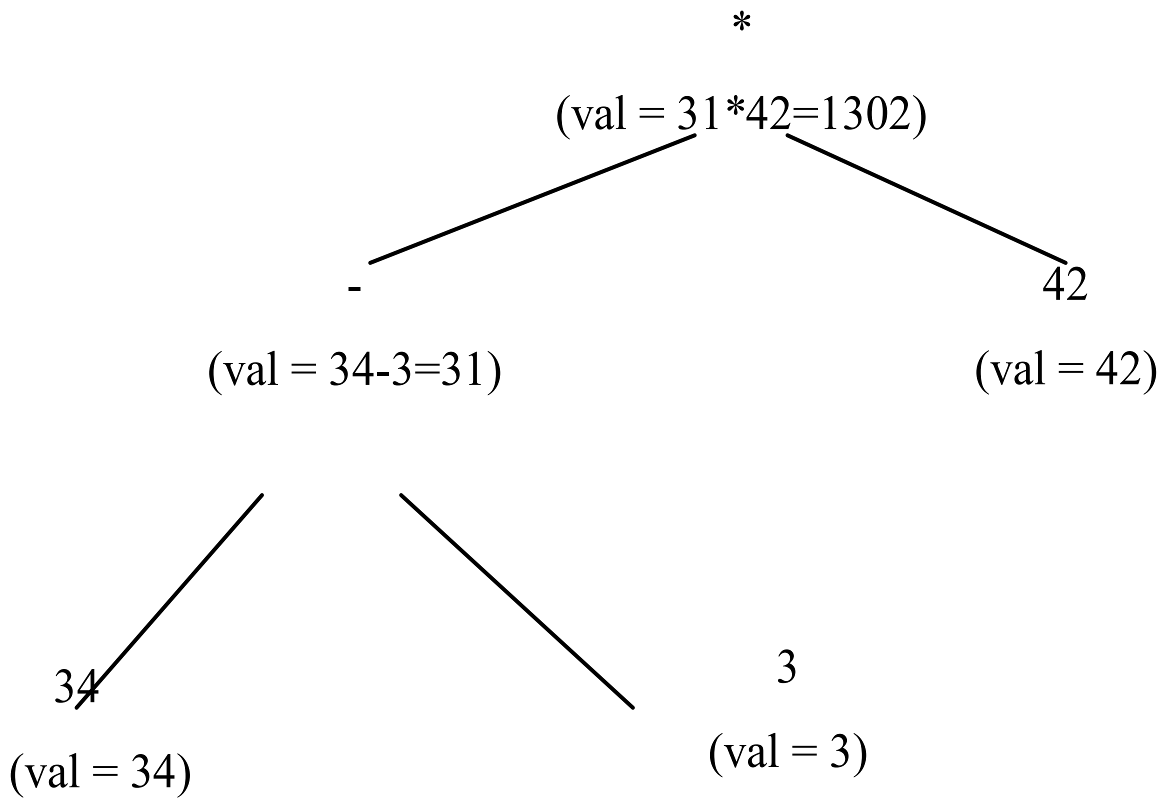

Fig 6.5

Example 6.5 define an abstract syntax tree for simple integer arithmetic expressions by the attribute grammar as follows:

Grammar Rule	Semantic Rules
exp1→exp2+term	exp1.tree=mkOpNode(+,exp2.tree,term.tree)
exp1→exp2-term	exp1.tree=mkOpNode(-,exp2.tree,term.tree)
exp1→ term	exp1.tree= term.tree
term1→term2*factor	term1.tree=mkOpNode(*,term2.tree,factor.tree)
term→factor	term.tree=factor.tree
factor→(exp)	factor.tree=exp.tree
factor→number	factor.tree=mkNumNode(number.lexval)

table 6.6

- Syntax tree itself is specified by an attribute grammar.
  - mkOpNode: construct a new tree node whose operator label is the first parameter, whose children are the second and third parameter.
  - mkNumNode: constructs a leaf node representing a number with that value.
- One question that is central to the specification of attributes using attribute grammars is :
  - How can we be sure that a particular attribute grammar is consistent and complete?
  - **That is, that it uniquely defines the give attributes?**
  - (the simple answer is that so far we cannot.)

End of Part One

THANKS