



# Programming Languages

## Lecture08 – Semantic Analysis



남 범 석

[bnam@skku.edu](mailto:bnam@skku.edu)

# Beyond Syntax

- There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
  int a, b, c, d;
  { ... }

fee() {
  int f[3],g[0],
    h, i, j, k;
  char *p;

  fie(h,i,"ab",j, k);
  k = f * i + j;
  h = g[17];
  printf("<%s,%s>\n", p,q);
  p = 10;
}
```

What is wrong with this program?

*(let me count the ways ...)*

- declared g[0], used g[17]
- wrong number of args to fie()
- "ab" is not an int
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are "deeper than syntax"

- To generate code, we need to understand its meaning !



## Beyond Syntax

- To generate code, the compiler needs to answer many questions
  - Is “x” a scalar, an array, or a function? Is “x” declared?
  - Are there names that are not declared? Declared but not used?
  - Which declaration of “x” does each use reference?
  - Is the expression “x \* y + z” type-consistent?
  - In “a[i,j,k]”, does a have three dimensions?
  - Where can “z” be stored? (register, local, global, heap, static)
  - In “f ← 15”, how should 15 be represented?
  - How many arguments does “fie()” take? What about “printf ()” ?
  - Does “\*p” reference the result of a “malloc()” ?
  - Do “p” & “q” refer to the same memory location?
  - Is “x” defined before it is used?



## Beyond Syntax

- These questions are part of context-sensitive analysis
  - Answers depend on values, not parts of speech
  - Questions & answers involve non-local information
  - Answers may involve computation
- How can we answer these questions?
  - Use formal methods
    - Attribute grammars (attributed grammars)
  - Use ad-hoc techniques
    - Ad-hoc syntax-directed translation (action routines)

# Attribute Grammars

- What is an attribute grammar?
  - A context-free grammar augmented with a set of rules
  - Each symbol in the derivation has a set of values, or attributes
  - The rules specify how to compute a value for each attribute

*Example grammar*

Number	→	Sign List
Sign	→	+
		-
List	→	List Bit
		Bit
Bit	→	0
		1

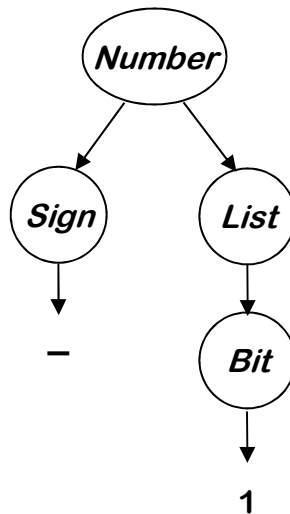
This grammar describes signed binary numbers

We would like to augment it with rules that compute the decimal value of each valid input string

# Examples

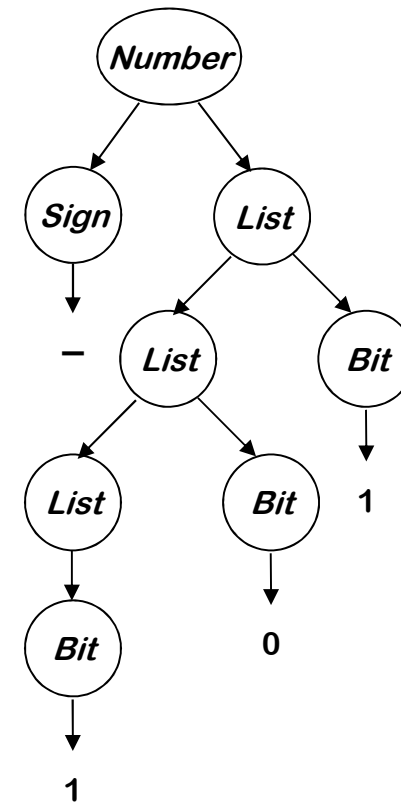
For “-1”

*Number* → *Sign List*  
 → - *List*  
 → - *Bit*  
 → - 1



For “-101”

*Number* → *Sign List*  
 → *Sign List Bit*  
 → *Sign List 1*  
 → *Sign List Bit 1*  
 → *Sign List 0 1*  
 → *Sign Bit 0 1*  
 → *Sign 1 0 1*  
 → - 101



*We will use these two throughout the lecture*

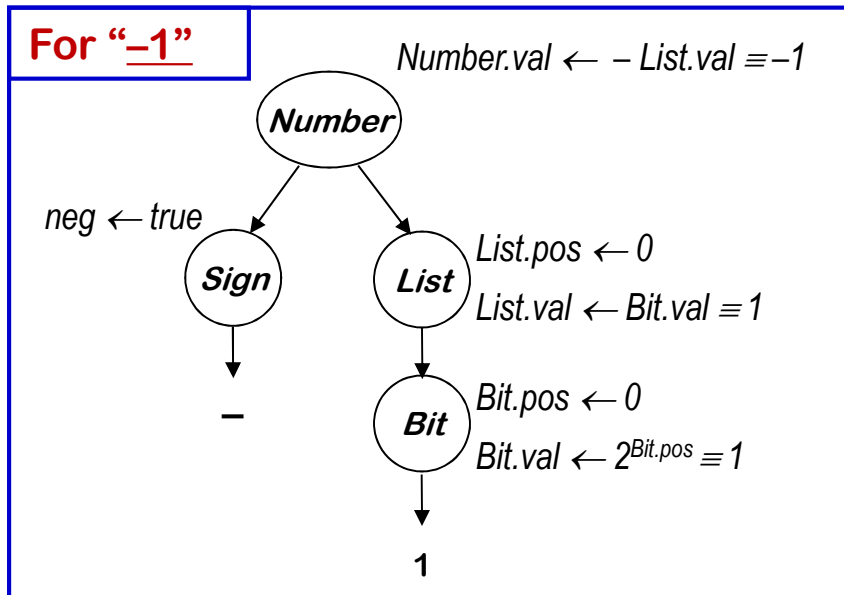
# Attribute Grammars

- Add rules to compute the decimal value of a signed binary number

<i>Productions</i>	<i>Attribution Rules</i>
<i>Number</i> → <i>Sign List</i>	$List.pos \leftarrow 0$ If <i>Sign.neg</i> then $Number.val \leftarrow - List.val$ else $Number.val \leftarrow List.val$
<i>Sign</i> → +	$Sign.neg \leftarrow false$
-	$Sign.neg \leftarrow true$
<i>List</i> <sub>0</sub> → <i>List</i> <sub>1</sub> <i>Bit</i>	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
<i>Bit</i>	$Bit.pos \leftarrow List.pos$ $List.val \leftarrow Bit.val$
<i>Bit</i> → 0	$Bit.val \leftarrow 0$
1	$Bit.val \leftarrow 2^{Bit.pos}$

Symbol	Attributes
<i>Number</i>	val
<i>Sign</i>	neg
<i>List</i>	pos, val
<i>Bit</i>	pos, val

# Back to the Examples



One possible evaluation order:

- 1 List.pos
- 2 Sign.neg
- 3 Bit.pos
- 4 Bit.val
- 5 List.val
- 6 Number.val

Other orders are possible

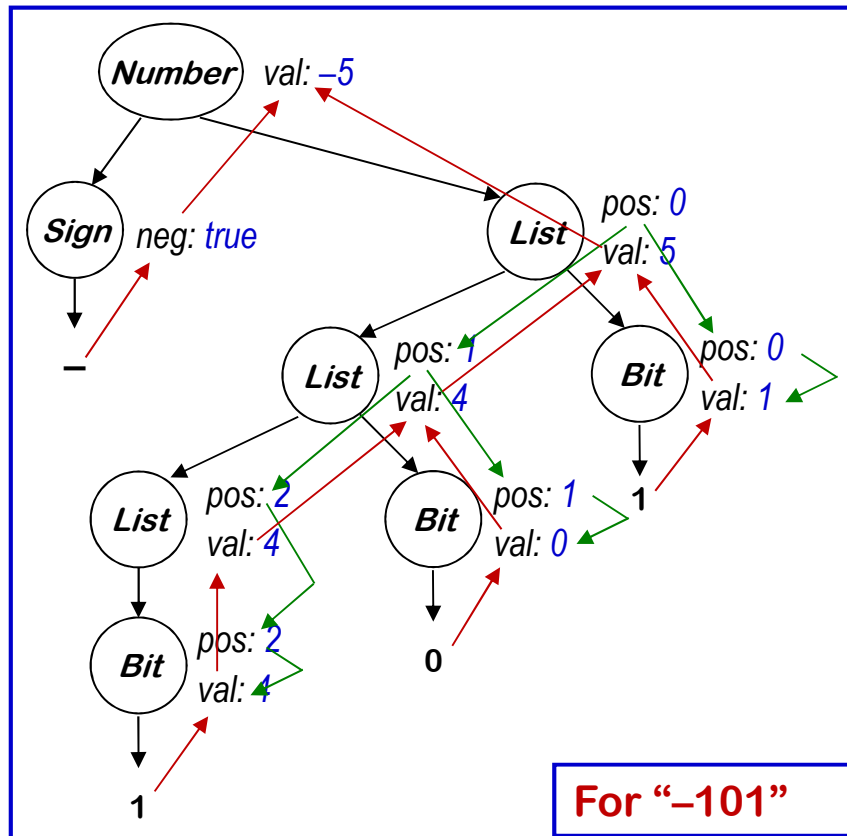
Knuth suggested a data-flow model for evaluation

- Independent attributes first
- Others in order as input values become available

Evaluation order must be consistent with the attribute dependence graph



# Back to the Examples



This is the complete attribute dependence graph for “-101”.

It shows the flow of *all* attribute values in the example.

Some flow downward  
→ inherited attributes

Some flow upward  
→ synthesized attributes

A rule may use attributes in the parent, children, or siblings of a node



# Attribute Grammar

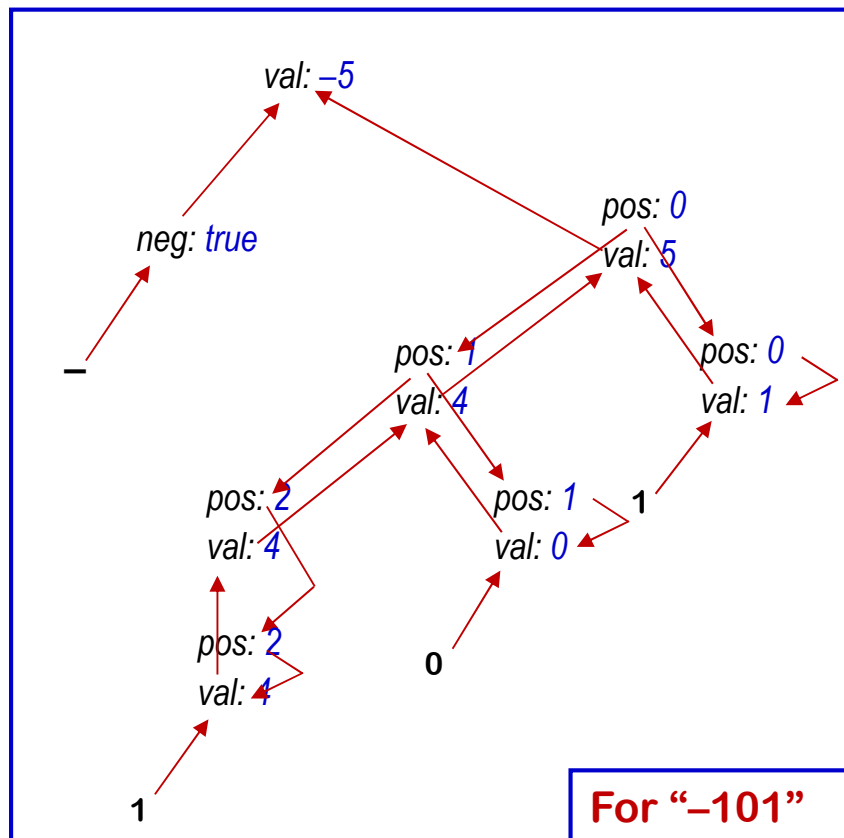
- The rules of game
  - Attributes associated with nodes in parse tree
  - Rules are value assignments associated with productions
  - Attribute is defined once, using local information
  - Label identical terms in production for uniqueness
  - Rules & parse tree define an attribute dependence graph
  
- Attributes fall into two classes: Synthesized and Inherited
  - Synthesized attribute
    - computed from the values of children nodes
  - Inherited attribute
    - computed from the sibling and parent nodes



# Attribute Evaluation Methods

- **Dynamic, dependence-based methods (dataflow)**
  - Build the parse tree
  - Build the dependence graph
  - Topological sort the dependence graph (circular dep. could fail)
  - Define attributes in topological order
- **Rule-based methods (treewalk)**
  - Analyze rules at compiler-generation time
  - Determine a fixed (static) ordering
  - Evaluate nodes in that order
- **Oblivious methods (left-to-right, right-to-left passes)**
  - Evaluation order is independent of rules & parse tree
  - Pick a convenient order (at design time) & use it
  - Restrict the AG that can be implemented

# Back to the Example



All that is left is the attribute dependence graph.

This succinctly represents the flow of values in the problem instance.

The **dynamic methods** sort this graph to find independent values, then work along graph edges.

The **rule-based methods** try to discover "good" orders by analyzing the rules.

The **oblivious methods** ignore the structure of this graph.

The dependence graph **must** be acyclic



# Circularity

- If a compiler uses attribute grammars, it must handle circularity.
- Avoidance
  - We can prove that some grammars can only generate instances with acyclic dependence graphs
  - S-attributed grammar has only synthesized attributes
    - No cycle in attribute dependence graphs
  - Largest such class is “strongly non-circular” grammars (SNC )
    - SNC grammars can be tested in polynomial time
- Evaluation
  - Iterative method works, if fixed-point problem



## Attribute Grammar Summary

- The attribute grammar formalism is important
  - Succinctly makes many points clear
  - Sets the stage for actual, ad-hoc practice
- The problems with attribute grammars
  - Information required by computations may be non local
  - Need for centralized information
- Some folks still argue for attribute grammars
  - Simplicity is still attractive
  - If attributes flow in a single direction, evaluation might be efficient
  - Not popular in real compilers

# Syntax-Directed Translation (SDT)

- Syntax-directed translation
  - a method of compiler implementation where the source language translation is completely driven by the parser
  - translating a string into a sequence of actions by attaching one action to each rule of a grammar
- Ad-Hoc SDT
  - Associate a snippet of code with each production
  - At each reduction, the corresponding snippet runs
  - Allowing arbitrary code provides complete flexibility
    - Includes ability to do bad things
- To make this work
  - Need names for attributes of each symbol on LHS & RHS
    - Typically, one attribute passed through parser
    - Yacc introduced \$\$, \$1, \$2, ... \$n, left to right
  - Need an evaluation scheme
    - Postorder
    - Fits nicely into LR(1) parsing algorithm
    - \$1, \$2, ... \$n are stored in the LR(1) parser stack

# Building an Abstract Syntax Tree

- Assume constructors for each node
- Assume stack holds pointers to nodes
- Assume yacc syntax

<i>Goal</i>	→ <i>Expr</i>	\$\$ = \$1;
<i>Expr</i>	→ <i>Expr</i> + <i>Term</i>	\$\$ = MakeAddNode(\$1,\$3);
	<i>Expr</i> - <i>Term</i>	\$\$ = MakeSubNode(\$1,\$3);
	<i>Term</i>	\$\$ = \$1;
<i>Term</i>	→ <i>Term</i> * <i>Factor</i>	\$\$ = MakeMulNode(\$1,\$3);
	<i>Term</i> / <i>Factor</i>	\$\$ = MakeDivNode(\$1,\$3);
	<i>Factor</i>	\$\$ = \$1;
<i>Factor</i>	→ ( <i>Expr</i> )	\$\$ = \$2;
	<u>number</u>	\$\$ = MakeNumNode(token);
	<u>id</u>	\$\$ = MakeIdNode(token);





## Reality

- Most parsers are based on this ad-hoc style of context-sensitive analysis
- Advantages
  - Addresses the shortcomings of the AG paradigm
  - Efficient, flexible
- Disadvantages
  - Must write the code with little assistance
  - Programmer deals directly with the details
- Most parser generators support a yacc-like notation

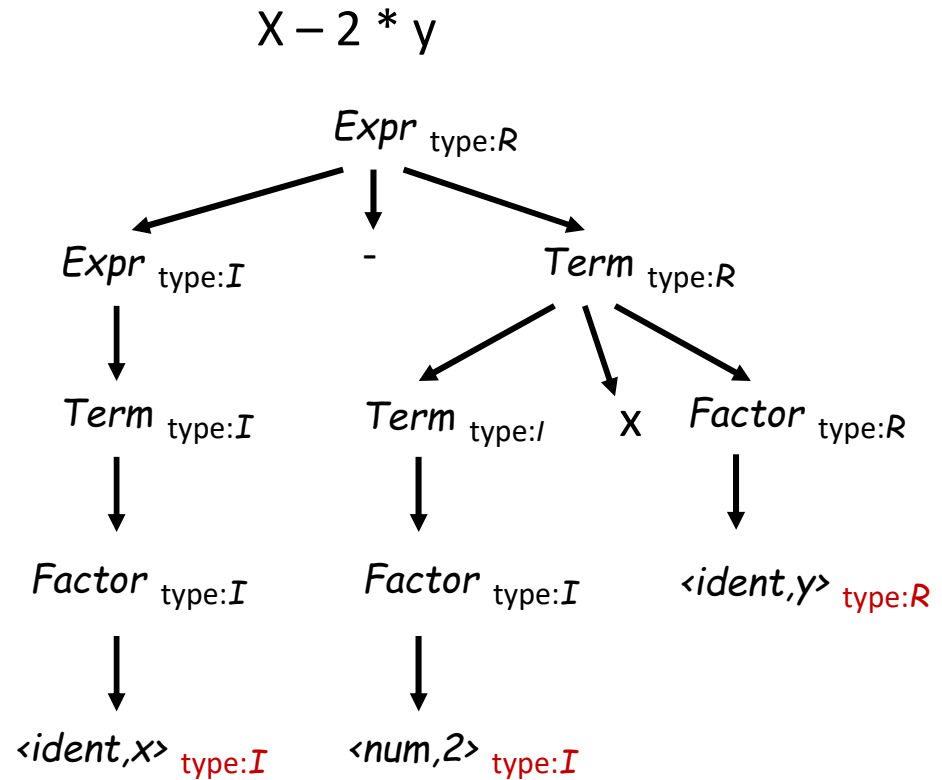
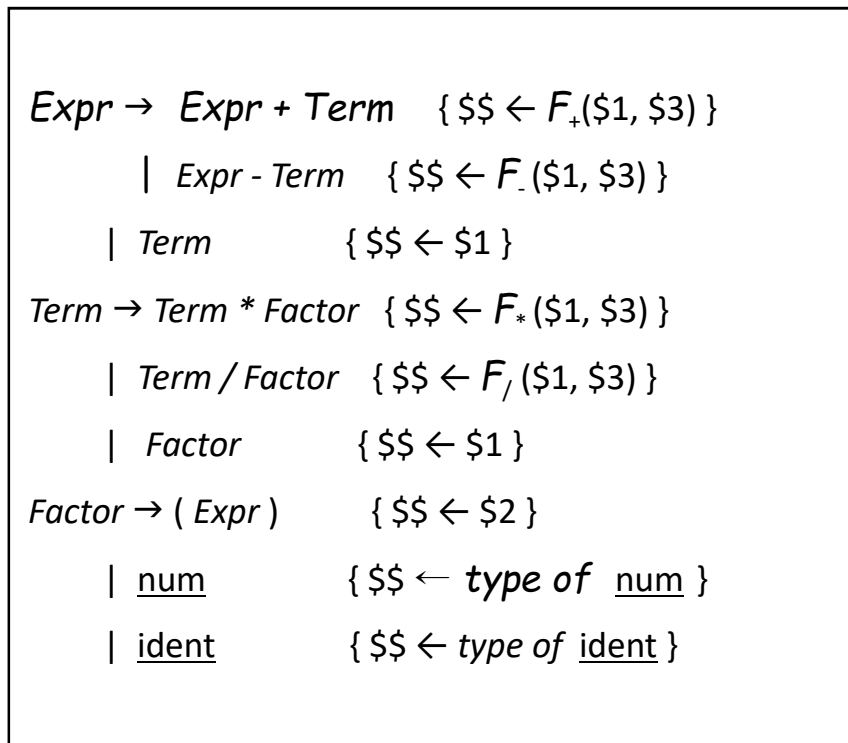


# Symbol Table

- Building a symbol table
  - Enter declaration information as processed
  - At end of declaration syntax, do some post processing
  - Use table to check errors as parsing progresses
- Simple error checking/type checking
  - Define before use → lookup on reference
  - Dimension, type, ... → check as encountered
  - Type conformability of expression → bottom-up walk
  - Procedure interfaces are harder

# Type Inference

- $F_+$ ,  $F_-$ ,  $F_*$ ,  $F_/\mathcal{}$  are result type mapping functions





# Summary

- **Attribute Grammars**
  - Pros: Formal, powerful, can deal with propagation strategies
  - Cons: Too many copy rules, no global tables, works on parse tree
  
- **Ad-hoc SDT (Postorder Code Execution)**
  - Pros: Simple and functional, can be specified in grammar (Yacc) but does not require parse tree
  - Cons: Rigid evaluation order, no context inheritance