Programming Language & Compiler

Context-Sensitive Analysis

Hwansoo Han

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 <u>int</u>
- 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	→	<u>+</u>
	|	<u>-</u>
List	→	List Bit
	|	Bit
Bit	→	0
	|	1

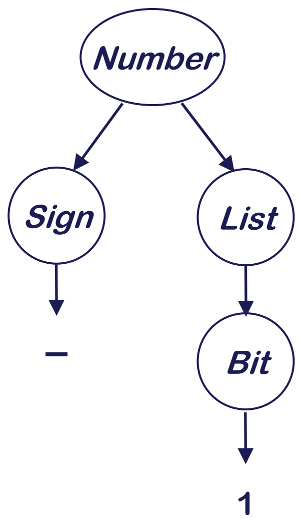
This grammar describes signed binary numbers

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

Examples

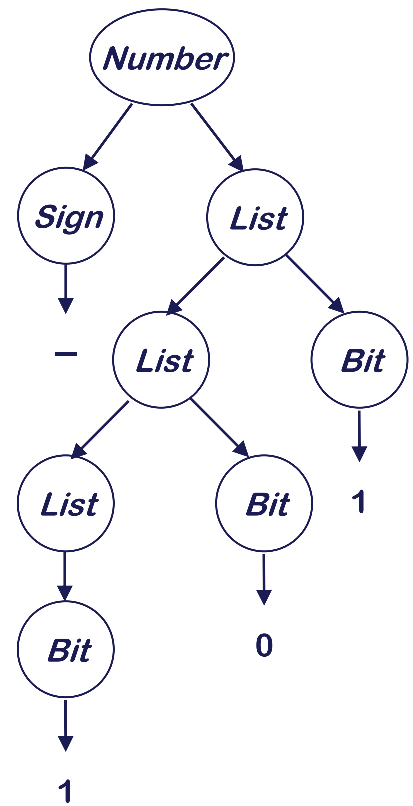
For "–1"

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

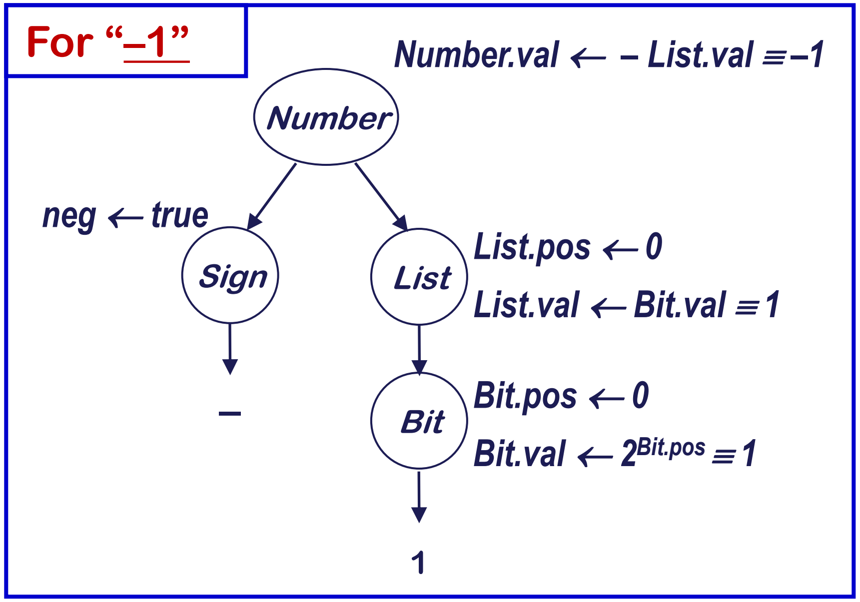
Attribute Grammars

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

Productions			Attribution Rules
Number	$\rightarrow$	*Sign List*	$List.pos \leftarrow 0$ If $Sign.neg$ then $Number.val \leftarrow -List.val$ else $Number.val \leftarrow List.val$
Sign	$\rightarrow$	$\underline{+}$	$Sign.neg \leftarrow false$
	|	$\underline{-}$	$Sign.neg \leftarrow true$
$List_0$	$\rightarrow$	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	$Bit.pos \leftarrow List.pos$ $List.val \leftarrow Bit.val$
Bit	$\rightarrow$	0	$Bit.val \leftarrow 0$
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

Back to the Examples



One possible evaluation order:

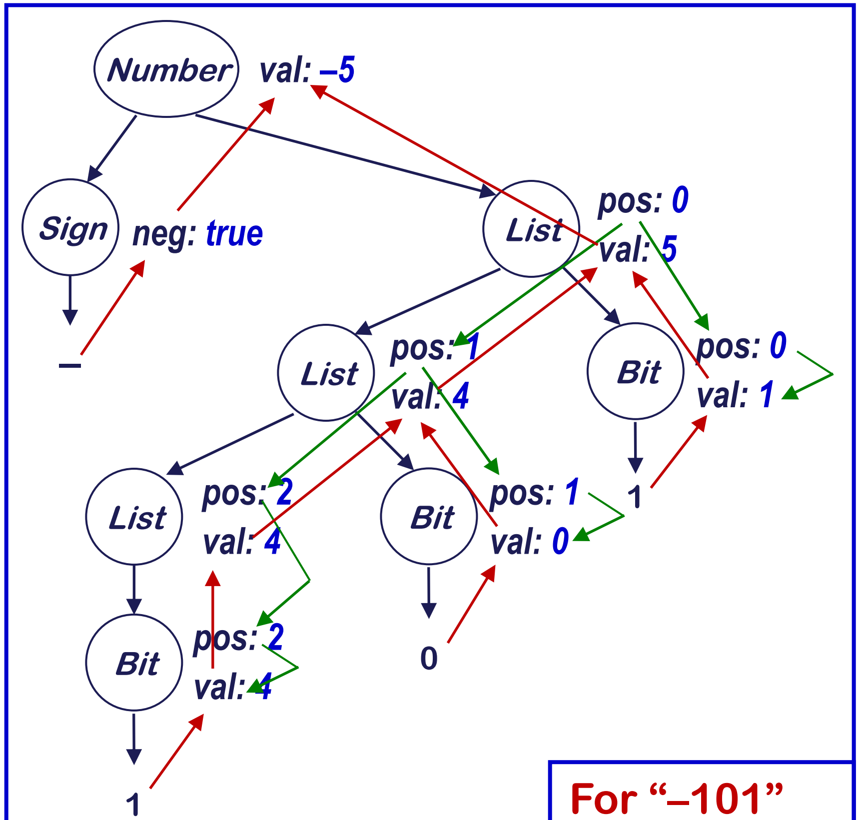
1 List.pos
2 Sign.neg
3 Bit.pos
4 Bit.val
5 List.val
6 Number.val

Other orders are possible

Evaluation order must be consistent with the attribute dependence graph

- **Knuth suggested a data-flow model for evaluation**
  - Independent attributes first
  - Others in order as input values become available

Back to the Examples



For "–101"

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

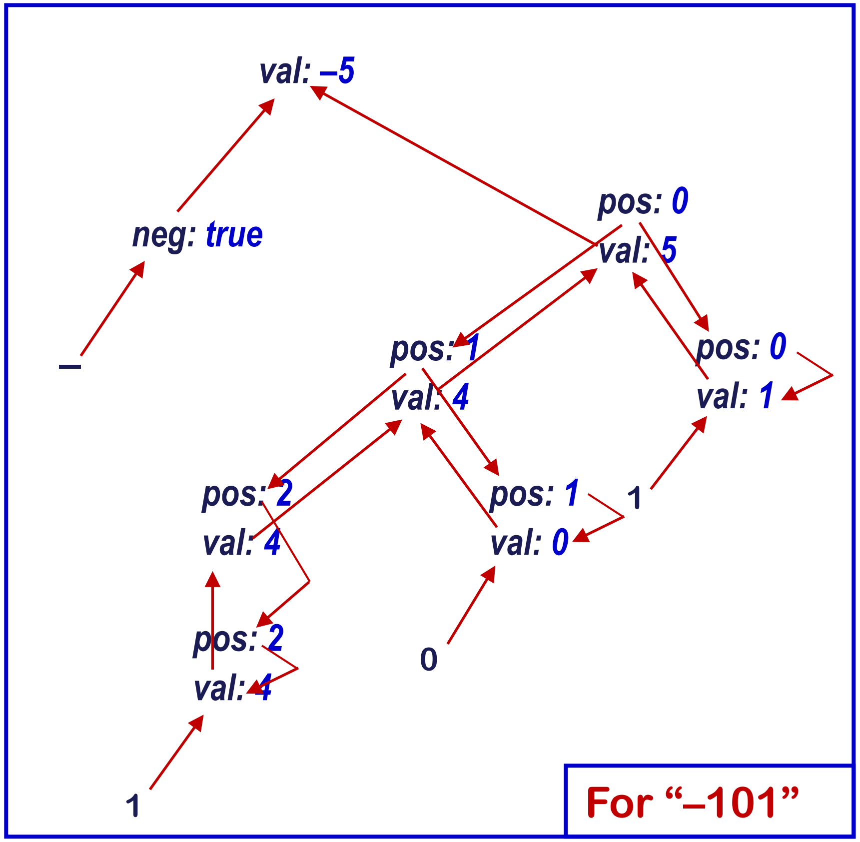
- **The rules of game**
  - Attributes associated with nodes in parse tree
  - Rules are value assignments associated with productions
  - Attribute is defined once, using local information
  - Label identical terms in production for uniqueness
  - Rules & parse tree define an attribute dependence graph

- **This produces a high-level, functional specification**
  - Synthesized attribute
    - Depends on values from children
  - Inherited attribute
    - Depends on values from siblings & parent

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
  - May 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**
  - Difficulty of <u>non-local computation</u>
  - 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

- ***Ad-hoc* syntax-directed translation**
  - 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 tasteless & 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**

Goal	→	*Expr*	$$ = $1;
Expr	→	*Expr* + *Term*	$$ = MakeAddNode($1,$3);
	|	*Expr* - *Term*	$$ = MakeSubNode($1,$3);
	|	*Term*	$$ = $1;
Term	→	*Term* * *Factor*	$$ = MakeMulNode($1,$3);
	|	*Term* / *Factor*	$$ = MakeDivNode($1,$3);
	|	*Factor*	$$ = $1;
Factor	→	(*Expr*)	$$ = $2;
	|	number	$$ = MakeNumNode(token);
	|	id	$$ = 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**

Typical Uses (symbol table)

- **Building a symbol table**
  - Enter declaration information as processed
    - TypeSpecifier, StorageClass, …
  - Do some context-sensitive analysis on a reduction
    - Number of StorageClass specifier
    - Validity of TypeSpecifier combination
  - Use table to check errors as parsing progresses

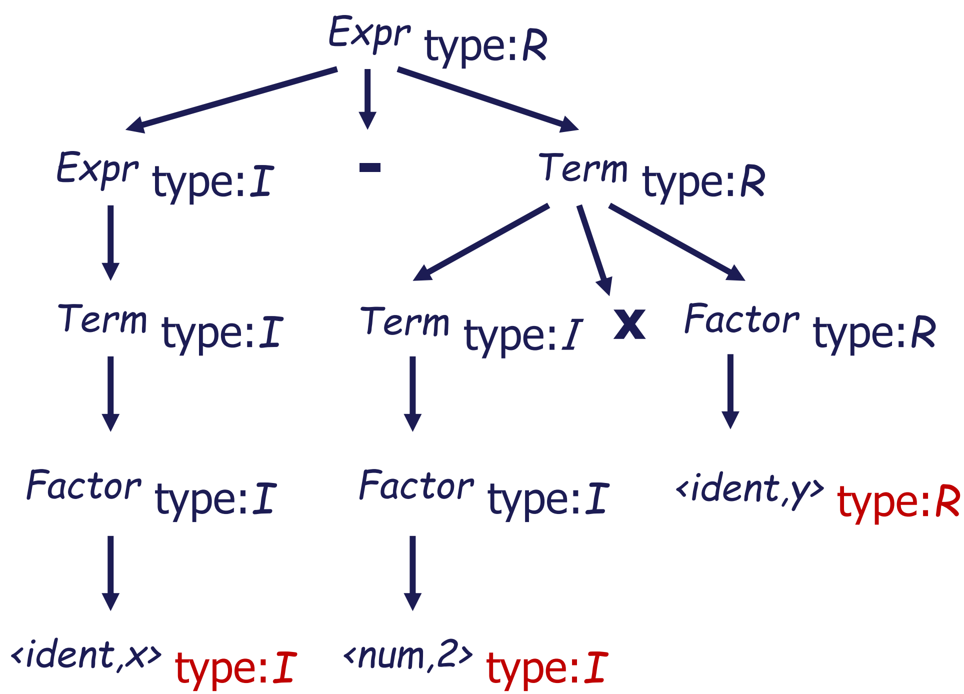
- **Simple error checking/type checking**
  - Define before use $\rightarrow$ lookup on reference
  - Dimension, type, ... $\rightarrow$ check as encountered
  - Type conformability of expression $\rightarrow$ bottom-up walk
  - Procedure interfaces are harder
    - Build a representation for parameter list & types
    - Create list of sites to check
    - Check offline, or handle the cases for arbitrary orderings

Typical Uses (type inference)

- F_+, F_-, F_*, $F_/$ **are result type mapping functions**

Expr →	*Expr* + *Term*		{ $\$\$ \leftarrow F_+(\$1, \$3)$ }
	| *Expr* - *Term*		{ $\$\$ \leftarrow F_-(\$1, \$3)$ }
	| *Term*		{ $\$\$ \leftarrow \$1$ }
Term →	*Term* * *Factor*		{ $\$\$ \leftarrow F_*(\$1, \$3)$ }
	| *Term* / *Factor*		{ $\$\$ \leftarrow F_/(\$1, \$3)$ }
	| *Factor*		{ $\$\$ \leftarrow \$1$ }
Factor →	(*Expr*)		{ $\$\$ \leftarrow \$2$ }
	| num		{ $\$\$ \leftarrow$ *type of* num }
	| ident		{ $\$\$ \leftarrow$ *type of* ident }

X – 2 * y



Limitations of Ad-hoc SDT (1)

- **Forced to evaluate in a given order: *postorder***
  - Left to right only
  - Bottom up only

- **Implications**
  - Declarations before uses
  - Context information cannot be passed down
    - How do you know what rule you are called from within?
    - Example: cannot pass bit position downwards
  - Could you use globals?
    - Requires initialization & some re-thinking of the solution
  - Can we rewrite it in a form that is better for the ad-hoc solution

Limitations of Ad-hoc SDT (2)

- **What about a rule that must work in mid-production?**
  - Can transform the grammar
    - Split it into two parts at the point where rule must go
    - Apply the rule on reduction to the appropriate part
  - Can also handle reductions on shift actions
    - Add a production to create a reduction

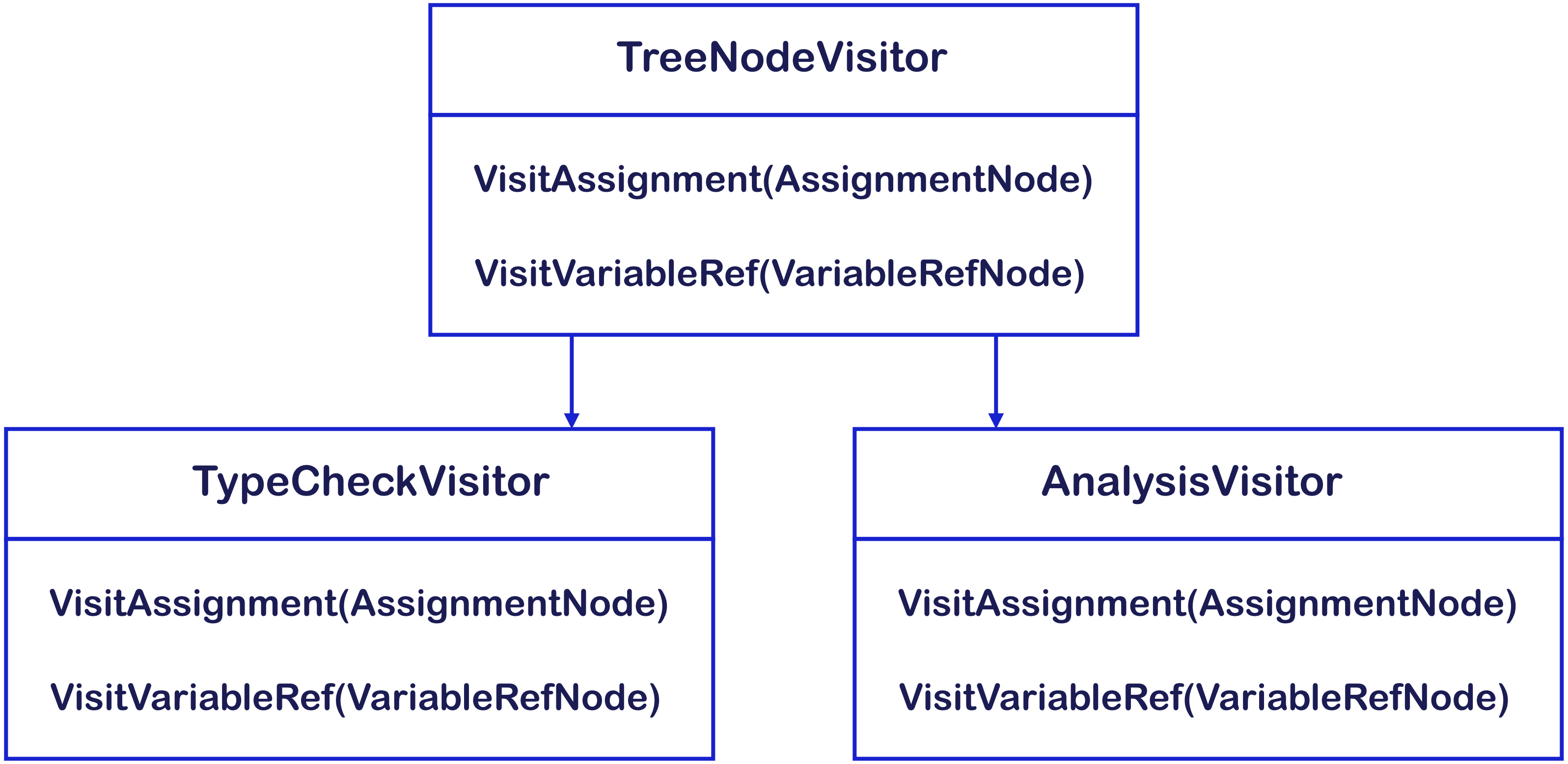
Was: $fee \rightarrow \underline{fum}$

Make it: $fee \rightarrow fie \rightarrow \underline{fum}$ and tie action to this reduction

- **Together, these let us apply rule at any point in the parse**

Alternative Strategy - treewalk

- **Build an abstract syntax tree**
  - Use tree walk routines
  - Use “visitor“ design pattern to add functionality



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

- **Generalized Tree Walk**
  - Pros: Full power and generality, operates on abstract syntax tree (using Visitor pattern)
  - Cons: Requires specific code for each tree node type, more complicated