

CS3300 - Compiler Design

Syntax Directed Translation

V. Krishna Nandivada

IIT Madras

- Attach rules or program fragments to productions in a grammar.
- Syntax directed definition (SDD)
 - $E_1 \rightarrow E_2 + T \quad E_1.code = E_2.code || T.code || '+'$
- Syntax directed translation Scheme (SDT)
 - $E \rightarrow E + T \quad \{\text{print '+'}\} // \text{semantic action}$
 - $F \rightarrow id \quad \{\text{print } id.val\}$



SDD and SDT scheme

- SDD: Specifies the values of attributes by associating semantic rules with the productions.
- SDT scheme: embeds program fragments (also called semantic actions) within production bodies.
 - The position of the action defines the order in which the action is executed (in the middle of production or end).
- SDD is easier to read; easy for specification.
- SDT scheme – can be more efficient; easy for implementation.



Example: SDD vs SDT scheme – infix to postfix trans

<i>SDTScheme</i>		<i>SDD</i>	
$E \rightarrow E + T$	$\{\text{print '+'}\}$	$E \rightarrow E + T$	$E.code = E.code T.code '+'$
$E \rightarrow E - T$	$\{\text{print '-'}\}$	$E \rightarrow E - T$	$E.code = E.code T.code '-'$
$E \rightarrow T$		$E \rightarrow T$	$E.code = T.code$
$T \rightarrow 0$	$\{\text{print '0'}\}$	$T \rightarrow 0$	$T.code = '0'$
$T \rightarrow 1$	$\{\text{print '1'}\}$	$T \rightarrow 1$	$T.code = '1'$
...		...	
$T \rightarrow 9$	$\{\text{print '9'}\}$	$T \rightarrow 9$	$T.code = '9'$



Syntax directed translation - overview

- 1 Construct a parse tree
- 2 Compute the values of the attributes at the nodes of the tree by visiting the tree

Key: We don't need to build a parse tree all the time.

- Translation can be done during parsing.
 - class of SDTs called "L-attributed translations".
 - class of SDTs called "S-attributed translations".



Syntax directed definition

- SDD is a CFG along with attributes and rules.
- An attribute is associated with grammar symbols (attribute grammar).
- Rules are associated with productions.



Attributes

- Attribute is any quantity associated with a programming construct.
- Example: data types, line numbers, instruction details

Two kinds of attributes: for a non-terminal A , at a parse tree node N

- A synthesized attribute: defined by a semantic rule associated with the production at N .

defined only in terms of attribute values at the children of N and at N itself.

- An inherited attribute: defined by a semantic rule associated with the parent production of N .

defined only in terms of attribute values at the parent of N siblings of N and at N itself.



Specifying the actions: Attribute grammars

Idea: attribute the syntax tree

- can add attributes (*fields*) to each node
- specify equations to define values (unique)
- can use attributes from parent and children

Example: to ensure that constants are immutable:

- add *type* and *class* attributes to expression nodes
- rules for production on $:=$ that
 - 1 check that LHS.*class* is *variable*
 - 2 check that LHS.*type* and RHS.*type* are consistent or conform



Attribute grammars

To formalize such systems Knuth introduced *attribute grammars*:

- grammar-based specification of tree attributes
- value assignments associated with productions
- each attribute uniquely, locally defined
- label identical terms uniquely

Can specify context-sensitive actions with attribute grammars



Example

PRODUCTION	SEMANTIC RULES
$D \rightarrow T L$	$L.in := T.type$
$T \rightarrow \mathbf{int}$	$T.type := \text{integer}$
$T \rightarrow \mathbf{real}$	$T.type := \text{real}$
$L \rightarrow L_1, \mathbf{id}$	$L_1.in := L.in$ $\text{addtype}(\mathbf{id}.entry, L.in)$
$L \rightarrow \mathbf{id}$	$\text{addtype}(\mathbf{id}.entry, L.in)$



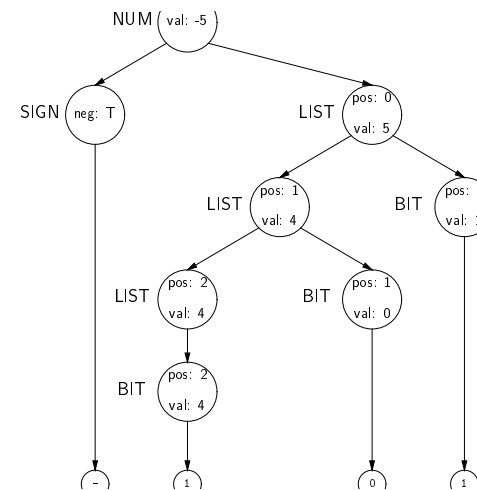
Example: Evaluate signed binary numbers

PRODUCTION	SEMANTIC RULES
$\text{NUM} \rightarrow \text{SIGN LIST}$	$\text{LIST.pos} := 0$ if SIGN.neg $\text{NUM.val} := -\text{LIST.val}$ else $\text{NUM.val} := \text{LIST.val}$
$\text{SIGN} \rightarrow +$	$\text{SIGN.neg} := \text{false}$
$\text{SIGN} \rightarrow -$	$\text{SIGN.neg} := \text{true}$
$\text{LIST} \rightarrow \text{BIT}$	$\text{BIT.pos} := \text{LIST.pos}$ $\text{LIST.val} := \text{BIT.val}$
$\text{LIST} \rightarrow \text{LIST}_1 \text{ BIT}$	$\text{LIST}_1.pos := \text{LIST.pos} + 1$ $\text{BIT.pos} := \text{LIST.pos}$ $\text{LIST.val} := \text{LIST}_1.val + \text{BIT.val}$
$\text{BIT} \rightarrow 0$	$\text{BIT.val} := 0$
$\text{BIT} \rightarrow 1$	$\text{BIT.val} := 2^{\text{BIT.pos}}$



Example (continued)

The attributed parse tree for -101:



- *val* and *neg* are synthesized attributes
- *pos* is an inherited attribute



Dependences between attributes

- values are computed from constants & other attributes
- *synthesized attribute* – value computed from children
- *inherited attribute* – value computed from siblings & parent
- *key notion*: induced dependency graph



The attribute dependency graph

- nodes represent attributes
- edges represent flow of values
- graph is specific to parse tree
- size is related to parse tree's size
- can be built alongside parse tree

The dependency graph must be acyclic

Evaluation order:

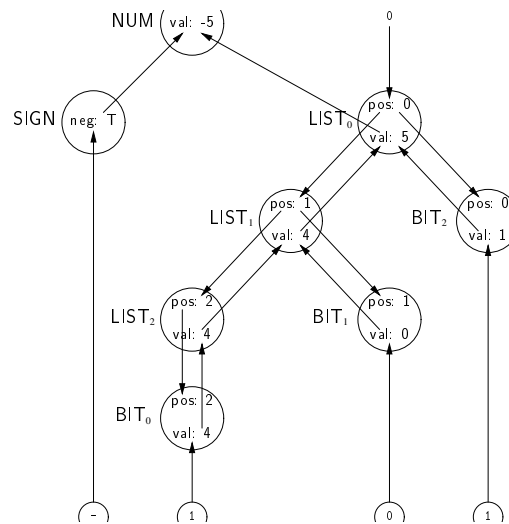
- topological sort the dependency graph to order attributes
- using this order, evaluate the rules

The order depends on both the grammar and the input string



Example (continued)

The attribute dependency graph:



Example: A topological order

- 1 SIGN.neg
- 2 LIST₀.pos
- 3 LIST₁.pos
- 4 LIST₂.pos
- 5 BIT₀.pos
- 6 BIT₁.pos
- 7 BIT₂.pos
- 8 BIT₀.val
- 9 LIST₂.val
- 10 BIT₁.val
- 11 LIST₁.val
- 12 BIT₂.val
- 13 LIST₀.val
- 14 NUM.val

Evaluating in this order yields NUM.val: -5



- *Parse-tree methods*

(dynamic)

- 1 build the parse tree
- 2 build the dependency graph
- 3 topological sort the graph
- 4 evaluate it

(cyclic graph fails)

What if there are cycles?



- Hard to tell, for a given grammar, whether there exists any parse tree whose dependency graphs have cycles.
- Focus on classes of SDD's that guarantee an evaluation order – do not permit dependency graphs with cycles.
 - L-attributed – class of SDTs called “L-attributed translations”.
 - S-attributed – class of SDTs called “S-attributed translations”.



Top-down (LL) on-the-fly one-pass evaluation

L-attributed grammar:

Informally – dependency-graph edges may go from left to right, not other way around.

given production $A \rightarrow X_1 X_2 \dots X_n$

- inherited attributes of X_j depend only on:
 - 1 inherited attributes of A
 - 2 arbitrary attributes of X_1, X_2, \dots, X_{j-1}
- synthesized attributes of A depend only on its inherited attributes and arbitrary RHS attributes
- synthesized attributes of an action depends only on its inherited attributes

i.e., evaluation order:

$\text{Inh}(A), \text{Inh}(X_1), \text{Syn}(X_1), \dots, \text{Inh}(X_n), \text{Syn}(X_n), \text{Syn}(A)$

This is precisely the order of evaluation for an LL parser



Bottom-up (LR) on-the-fly one-pass evaluation

S-attributed grammar:

- L-attributed
- only synthesized attributes for non-terminals
- actions at far right of a RHS

Can evaluate S-attributed in one bottom-up (LR) pass.



Evaluate S-attributed grammar in bottom-up parsing

- Evaluate it in any bottom-up order of the nodes in the parse tree.
- (One option:) Apply *postorder* to the root of the parse tree:

```
void postorder (N) {
    for (each child C of N)
        do
            postorder(C);
    evaluate the attributes associated with N;
done
}
```

- post order traversal of the parse tree corresponds to the exact order in which the bottom-up parsing builds the parse tree.
- Thus, we can evaluate S-attributed in one bottom-up (LR) pass.



Inherited Vs Synthesised attributes

Synthesized attributes are limited

Inherited attributes (are good): derive values from constants, parents, siblings

- used to express context (*context-sensitive checking*)
- inherited attributes are more “natural”

We want to use both kinds of attributes

- can *always* rewrite L-attributed LL grammars (using markers and copying) to avoid inherited attribute problems with LR

Self reading (if interested) – Dragon book Section 5.5.4.



LL parsers and actions

How does an LL parser handle (aka - execute) actions?

Expand productions *before* scanning RHS symbols, so:

- push actions onto parse stack like other grammar symbols
- pop and perform action when it comes to top of parse stack



LL parsers and actions

```
push EOF
push Start Symbol
token ← next_token()
repeat
    pop X
    if X is a terminal or EOF then
        if X = token then
            token ← next_token()
        else error()
    else if X is an action
        perform X
    else /* X is a non-terminal */
        if M[X,token] = X → Y1Y2...Yk then
            push Yk, Yk-1, ..., Y1
        else error()
until X = EOF
```



LR parsers and action symbols

What about LR parsers?

Scan entire RHS before applying production, so:

- cannot perform actions until entire RHS scanned
- can only place actions at very end of RHS of production
- introduce new marker non-terminals and corresponding productions to get around this restriction[†]

$$A \rightarrow w \text{ action } \beta$$

becomes

$$A \rightarrow M\beta$$

$$M \rightarrow w \text{ action}$$

[†]yacc, bison, CUP do this automatically



Action-controlled semantic stacks

- Approach:
 - stack is managed explicitly by action routines
 - actions take arguments from top of stack
 - actions place results back on stack
- Advantages:
 - actions can directly access entries in stack without popping (efficient)
- Disadvantages:
 - implementation is exposed
 - action routines must include explicit code to manage stack (or use `stack` abstract data type).



LR parser-controlled semantic stacks

Idea: let parser manage the semantic stack

LR parser-controlled semantic stacks:

- parse stack contains already parsed symbols
 - maintain semantic values in parallel with their symbols
 - add space in parse stack or parallel stack for semantic values
 - every matched grammar symbol has semantic value
 - pop semantic values along with symbols
- ⇒ LR parsers have a very nice fit with semantic processing



LL parser-controlled semantic stacks

Problems:

- parse stack contains predicted symbols, not yet matched
- often need semantic value after its corresponding symbol is popped

Solution:

- use separate semantic stack
- push entries on semantic stack along with their symbols
- on completion of production, pop its RHS's semantic values



Advantages

- clean formalism
- automatic generation of evaluator
- high-level specification

Disadvantages

- evaluation strategy determines efficiency
- increased space requirements
- parse tree evaluators need dependency graph
- results distributed over tree
- circularity testing

Intel's 80286 Pascal compiler used an attribute grammar evaluator to perform context-sensitive analysis.

Historically, attribute grammar evaluators have been deemed too large and expensive for commercial-quality compilers.

