Syntax Directed Translation

Syntax-Directed Translation

- Attach rules or program fragments to productions in a grammar.
- The compilation process is guided by context-free grammars.
  - Symbol table generation, type-checking, intermediate code generation, etc. are all carried out by syntax-directed translations.
- The main idea is to associate *attributes* with grammar symbols.

- Two ways to perform syntax-directed translations:
  1. Syntax directed definition (SDD)
     - $E_1 \rightarrow E_2 + T$ $\qquad E_1.code = E_2.code || T.code ||' +'$
  2. Syntax directed translation Scheme (SDT)
     - $E \rightarrow E + T$ $\qquad$ {print '+'} // semantic action
     - $F \rightarrow id$ $\qquad$ {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

SDTScheme		*SDD*	
$E \rightarrow E + T$	$\{print'+'\}$	$E \rightarrow E + T$	$E.code = E.code \| \| T.code \| \| '+'$
$E \rightarrow E - T$	$\{print'-'\}$	$E \rightarrow E - T$	$E.code = E.code \| \| T.code \| \| '-'$
$E \rightarrow T$		$E \rightarrow T$	$E.code = T.code$
$T \rightarrow 0$	$\{print'0'\}$	$T \rightarrow 0$	$T.code = '0'$
$T \rightarrow 1$	$\{print'1'\}$	$T \rightarrow 1$	$T.code = '1'$
$\dots$		$\dots$	
$T \rightarrow 9$	$\{print'9'\}$	$T \rightarrow 9$	$T.code = '9'$

Syntax directed translation - Parse tree

Idiomatic syntax directed translation does the following:

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.

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 parse 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 constant variables are immutable:

- add *type* (`int,bool,...`) and *kind* (`var, const`) attributes expression nodes.
- rules for production on `:=` (assignment) that
  1. check that LHS *kind* is `var`
  2. check that LHS *type* and RHS *type* are consistent or conform

Attribute grammars

Formally, we define the notion of *attribute grammars*:

- grammar-based specification of parse-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
$T \rightarrow$ **int**	T.type := integer
$T \rightarrow$ **real**	T.type := real
$L \rightarrow L_1$, **id**	L_1.in := L.in
	addtype(**id**.entry, L.in)
$L \rightarrow$ **id**	addtype(**id**.entry, L.in)

Example: Evaluate signed binary numbers

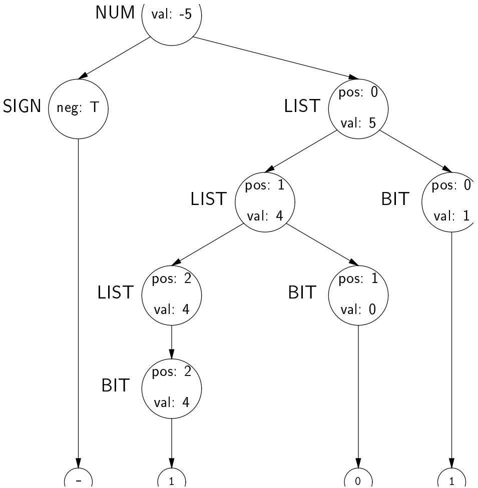
PRODUCTION	SEMANTIC RULES
NUM $\rightarrow$ SIGN LIST	
SIGN $\rightarrow$ `+`	
SIGN $\rightarrow$ `-`	
LIST $\rightarrow$ BIT	
LIST $\rightarrow$ LIST_1 BIT	
BIT $\rightarrow$ `0`	
BIT $\rightarrow$ `1`	

Example: Evaluate signed binary numbers

PRODUCTION	SEMANTIC RULES
NUM $\rightarrow$ SIGN LIST	LIST.pos := 0
	if SIGN.neg
	NUM.val := -LIST.val
	else
	NUM.val := LIST.val
SIGN $\rightarrow$ +	SIGN.neg := false
SIGN $\rightarrow$ -	SIGN.neg := true
LIST $\rightarrow$ BIT	BIT.pos := LIST.pos
	LIST.val := BIT.val
LIST $\rightarrow$ LIST_1 BIT	LIST_1.pos := LIST.pos + 1
	BIT.pos := LIST.pos
	LIST.val := LIST_1.val + BIT.val
BIT $\rightarrow$ 0	BIT.val := 0
BIT $\rightarrow$ 1	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

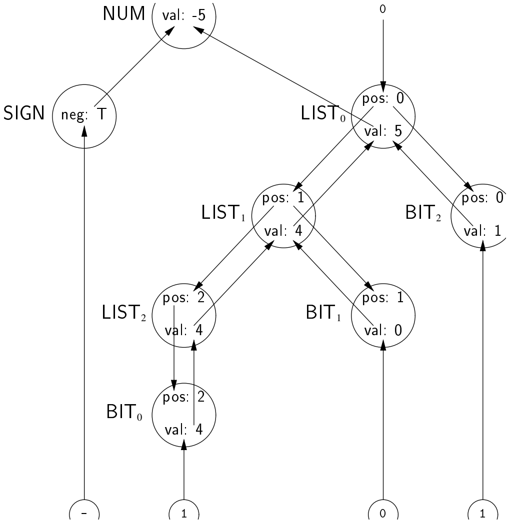
Evaluation order:

- 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. LIST_0.pos
3. LIST_1.pos
4. LIST_2.pos
5. BIT_0.pos
6. BIT_1.pos
7. BIT_2.pos
8. BIT_0.val
9. LIST_2.val
10. BIT_1.val
11. LIST_1.val
12. BIT_2.val
13. LIST_0.val
14. NUM.val

Evaluating in this order yields NUM.val: -5

Evaluation strategies

- *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?

Avoiding 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".

L-Attributed Grammars

Informally – allows both synthesized and inherited attributes, but dependency-graph edges may only go from left to right, not other way around.

Given production $A \to X_1 X_2 \cdots X_n$

- Synthesized attributes of A
- Inherited attributes of X_j depend only on:
  1. Inherited attributes of A
  2. Arbitrary attributes of $X_1, X_2, \cdots X_{j-1}$

i.e., evaluation order:
Inh(A), Inh(X_1), Syn(X_1), ..., Inh(X_n), Syn(X_n), Syn(A) This is precisely the order of evaluation for an LL parser

L-Attributed Grammar: Examples

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

L-Attributed Grammar: Examples

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

Evaluating attributes of L-attributed grammar

- Perform depth-first traversal starting from the root of the parse tree:

```
void  depth-first (N) {
    evaluate the inherited attributes of N;
    for (each child C of N in left-to-right order)
     do
      depth-first(C);
     done
    evaluate the synthesized attributes of N;
}
```

- Note that this order of visiting nodes corresponds to the exact order in which top-down parser builds the parse tree.
- Thus, we can also evaluate L-attributed grammars in one top-down (LL) pass.

SDT for L-Attributed Grammars

- Embed the action which evaluates an attribute inside the body of the production.
- The action for evaluating an inherited attribute for X is placed immediately before the occurrence of X in the body of the production.
- The action for evaluating a synthesized attribute for A is placed after the entire body of the production.

The SDT for $A \rightarrow X_1 X_2 \ldots X_n$ is
$A \rightarrow \{\textsc{Inh}(X_1) = \cdots\} X_1 \{\textsc{Inh}(X_2) = \cdots\} X_2 \ldots X_n \{\textsc{Syn}(\mathsf{A}) = \cdots\}$

S-attributed Grammars

- allows only synthesized attributes for non-terminals
- equivalently, semantic actions at far right of a RHS

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

Evaluating attributes of S-attributed grammar

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

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

- 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 grammars in one bottom-up (LR) pass.

LL parsers and actions

How can we directly evaluate attributes in a L-attributed SDT during LL parsing?

During LL Parsing, we 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 → Y1 Y2 ··· Yk then
            push Yk, Yk−1, ··· , Y1
        else error()
until X = EOF
```

The attribute values can be stored on the stack as well. For more details, refer to Dragon Book, Chapter 5, Section 5.5.3.

LR parsers and actions

What about LR parsers?
In LR Parsing, we 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

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

becomes

$$A \rightarrow M\beta$$

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

For more details, refer to Dragon Book, Chapter 5, Section 5.5.4

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) to avoid inherited attribute problems with LR