MCA Part III

Paper- XIX

Topic: Syntax Directed Translation

Prepared by: Dr. Kiran Pandey

School of Computer science

Email-Id: kiranpandey.nou@gmail.com

INTRODUCTION

Syntax Directed Translation means that an input sentence is related to its syntactic structure, i.e., to its Parse-Tree. By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars. The grammar symbols representing the language constructs is associated with attributes. Values for attributes are computed by Semantic Rules associated with grammar productions. Evaluation of Semantic Rules may:

(i) Generate Code;
(ii) Insert information into the Symbol Table;
(iii) Perform Semantic Check; – Issue error messages.

There are two notations for attaching semantic rules:

1. Syntax Directed Definitions. High-level specification hiding many implementation details (also called Attribute Grammars).

2. Translation Schemes. More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

SYNTAX DIRECTED TRANSLATION SCHEMES

Syntax Directed Definitions are a generalization of context-free grammars in which:

1. Grammar symbols have an associated set of Attributes;
2. Productions are associated with Semantic Rules for computing the values of attributes.

Such formalism generates Annotated Parse -Trees where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X). The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node. We distinguish between two kinds of attributes:

1. **Synthesized Attributes:** They are computed from the values of the attributes of the children nodes.

2. **Inherited Attributes:** They are computed from the values of the attributes of both the siblings and the parent nodes.

Definitions

Each production, $A \rightarrow \alpha$, is associated with a set of semantic rules: b: = f (c 1, c 2, . . . , c k), where f is a function and either 1. b is a synthesized attribute of A, and c 1, c 2, . . . , c k are attributes of the grammar symbols of the production, or 2. b is an inherited attribute of a grammar symbol in α, and c 1, c 2, . . . , c k are attributes of grammar symbols in α or attributes of A.

(**Note:** Terminal symbols are assumed to have synthesized attributes supplied by the lexical analyzer).

Procedure calls (e.g. print in the next slide) define values of Dummy synthesized attributes of the non-terminal on the left-hand side of the production.

Example

Example. Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called val.

PRODUCTION	SEMANTIC RULE
$L \rightarrow E\text{n}$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \text{digit}$	$F.val :=\text{digit}.lexval$

The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree. By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars. The grammar symbols representing the language constructs is associated with attributes. Values for attributes are computed by Semantic Rules associated with grammar productions.

Translation Schemes are more implementation oriented than syntax directed definitions since they indicate the order in which semantic rules and attributes are to be evaluated.

Definition: A Translation Scheme is a context-free grammar in which

1. Attributes are associated with grammar symbols;

2. Semantic Actions are enclosed between braces {} and are inserted within the right-hand side of productions.

Yacc uses Translation Schemes. Translation Schemes deal with both synthesized and inherited attributes. Semantic Actions are treated as terminal symbols:

Annotated parse-trees contain semantic actions as children of the node standing for the corresponding production. Translation Schemes are useful to evaluate L:

-Attributed definitions at parsing time (even if they are a general mechanism).

- An L-Attributed Syntax-Directed Definition can be turned into a Translation Scheme.

Example

Consider the Translation Scheme for the L-Attributed Definition for “type declarations”:

$$D \rightarrow T \ \{L.in := T.type\} \ L$$

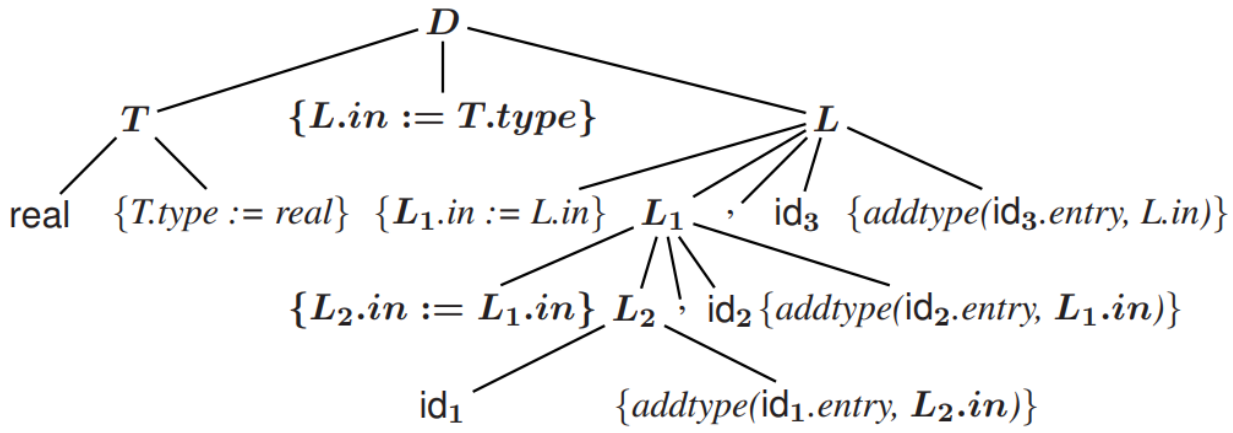
$$T \rightarrow \text{int} \ \{T.type := integer\}$$

$$T \rightarrow \text{real} \ \{T.type := real\}$$

$$L \rightarrow \{L_1.in := L.in\} \ L_1, \text{id} \ \{addtype(\text{id}.entry, L.in)\}$$

$$L \rightarrow \text{id} \ \{addtype(\text{id}.entry, L.in)\}$$

The parse-tree with semantic actions for the input real id 1, id 2, id 3 is:



Traversing the Parse-Tree in depth-first order (Post Order) we can evaluate the attributes.

Design of Translation Schemes

When designing a Translation Scheme we must be sure that an attribute value is available when a semantic action is executed. When the semantic action involves only synthesized attributes: The action can be put at the end of the production.

Example: The following Production and Semantic Rule:

$T \rightarrow T1 * F$ T .val := T1.val $*$ F.val

yield the translation scheme:

$T \rightarrow T1 * F$ {T .val := T1.val $*$ F.val }

Rules for Implementing L-Attributed SDD's.

If we have an L-Attributed Syntax-Directed Definition we must enforce the following restrictions:

1. An inherited attribute for a symbol in the right-hand side of a production must be computed in an action before the symbol;

2. A synthesized attribute for the non-terminal on the left-hand side can only be computed when all the attributes it references have been computed:

The action is usually put at the end of the production.

Compile-Time Evaluation of Translation Schemes

Attributes in a Translation Scheme following the above rules can be computed at compile time similarly to the evaluation of S-Attributed Definitions. Starting from a Translation Scheme (with embedded actions) we introduce a transformation that makes all the actions occur at the right ends of their productions.

– For each embedded semantic action we introduce a new Marker (i.e., a non-terminal, say M) with an empty production ($M \rightarrow \varrho$);

– The semantic action is attached at the end of the production $M \rightarrow \varrho$.

• Example. Consider the following translation scheme:

$S \rightarrow aA$ { C.i = f(A.s) } C

$S \rightarrow bAB$ { C.i = f(A.s) } C

$C \rightarrow c$ {C.s = g (C.i) }

Then, we add new markers M1, M2 with:

$S \rightarrow aA$ M 1 C

$S \rightarrow bAB$ M 2 C

M 1 → ϱ { M1.s := f (val[top]) }

M 2 → ϱ { M2.s := f (val[top − 1]) }

C → c {C.s := g (val[top − 1]) }

The inherited attribute of C is the synthesized attribute of either M1 or M2: The value of C.i is always in val [top -1] when C → c is applied. General rules to compute translations schemes during bottom-up parsing assuming an L-attributed grammar.

• For every production A → X 1 . . . X n introduce n new markers M1, . . . ,Mn and replace the production by A → M1 X 1 . . .Mn X n.

• Thus, we know the position of every synthesized and inherited attribute of Xj and A:

1. Xj.s is stored in the val entry in the parser stack associated with Xj ;

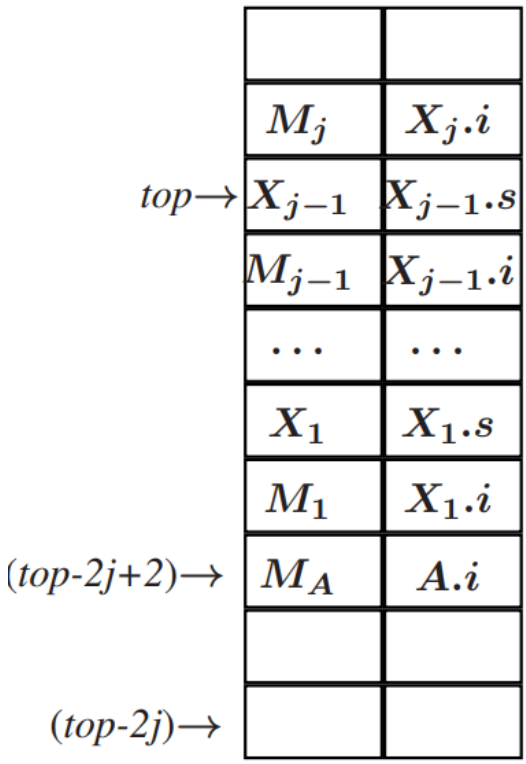
2. Xj.i is stored in the val entry in the parser stack associated with Mj ;

3. A.i is stored in the val entry in the parser stack immediately before the position storing M1.

Remark 1. Since there is only one production for each marker a grammar remains LL(1) with addition of markers.

• Remark 2. Adding markers to an LR(1) Grammar can introduce conflicts for not L-Attributed SDD's!!!

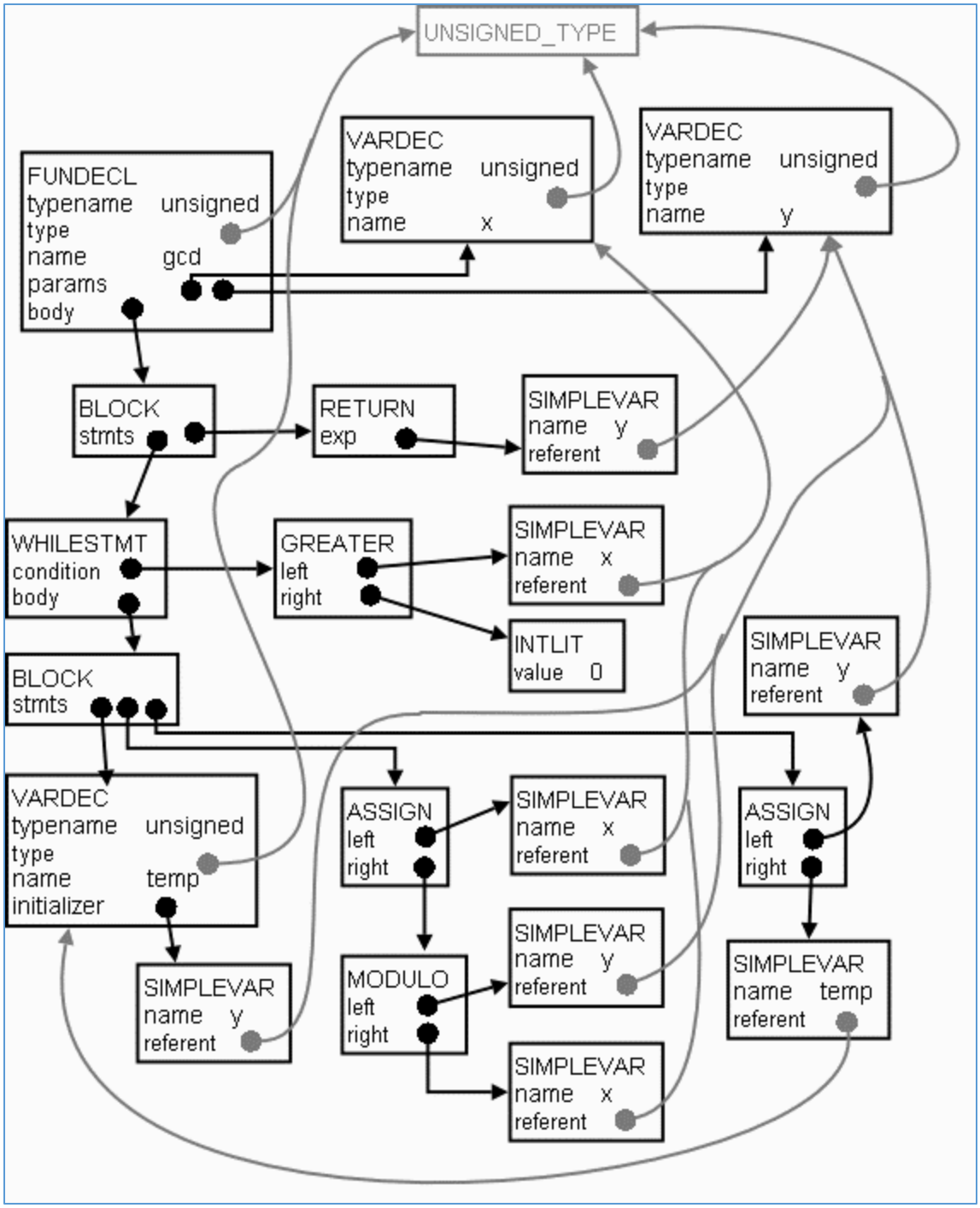
Example: Computing the inherited attribute Xj.i after reducing with Mj → ϱ.



- $A.i$ is in $val[top - 2j + 2]$;
- $X_1.i$ is in $val[top - 2j + 3]$;
- $X_1.s$ is in $val[top - 2j + 4]$;
- $X_2.i$ is in $val[top - 2j + 5]$;
- and so on.

OVERALL DESIGN OF A SEMANTIC ANALYSER

During semantic analysis we have to check legality rules and while doing so, we tie up the pieces of the syntax tree (by resolving identifier references, inserting cast operations for implicit coercions, etc.) to form a semantic graph. The concept is explained using a diagram given below:



Obviously, the set of legality rules **is different for each language**. Examples of legality rules you might see in a Java-like language include:

- Multiple declarations of a variable within a scope
- Referencing a variable before its declaration
- Referencing an identifier that has no declaration
- Violating access (public, private, protected, ...) rules

- Too many arguments in a method call
- Not enough arguments in a method call
- Type mismatches (there are tons of these)

Errors occurring during this phase are called static semantic errors.

Questions

1. Define syntax directed translation.
2. Give an example of semantic analyser.