THEORY OF COMPUTATION AND COMPILERS

Unit - III

SEMANTIC ANALYSIS, INTERMEDIATE CODE GENERATOR & SYMBOL TABLE

SEMANTIC ANALYSIS

- Attributed Grammars
- Syntax Directed Translation 

INTERMEDIATE CODE GENERATOR

- Intermediate Forms of Source Programs - Abstract Syntax Tree, Polish Notation and Three Address Codes
- Intermediate Code Forms
- Type Checker

SYMBOL TABLE

- Symbol Table Format
- Organization for Block Structures Languages
- Hashing

Dr. R. Madana Mohana
Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning
CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

Syntax-Directed Translation

Syntax-Directed Translation (SDT) Schemes

Syntax-Directed Translation (SDT) Schemes

- Postfix Translation Schemes
- Parser-Stack Implementation of Postfix SDT's
- SDT's With Actions Inside Productions

Syntax-Directed Translation (SDT) Schemes

Syntax-Directed Translation scheme (SDT) are a complementary notation to **Syntax-Directed Definitions (SDD)**. All of the applications of Syntax-Directed Definitions can be implemented using Syntax-Directed Translation schemes.

Definition:

A **Syntax-Directed Translation scheme (SDT)** is a context-free grammar with program fragments embedded within production bodies. The program fragments are called semantic actions and can appear at any position within a production body. By convention, we place curly braces around actions; if braces are needed as grammar symbols, then we quote them.

Syntax-Directed Translation (SDT) Schemes

- Any **SDT** can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal.
- Typically, **SDT**'s are implemented during parsing, without building a parse tree.
- The use of **SDT**'s to implement two important classes of **SDD**'s:
  1. The underlying grammar is **LR-parsable**, and the **SDD** is **S-attributed**.
  2. The underlying grammar is **LL-parsable**, and the **SDD** is **L-attributed**.

Syntax-Directed Translation (SDT) Schemes

- The semantic rules in an **SDD** can be converted into an **SDT** with actions that are executed at the right time. During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of the action have been matched.
- **SDT**'s that can be implemented during parsing can be characterized by introducing distinct *marker nonterminals* in place of each embedded action; each marker **M** has only one production, **M → ϵ** . If the grammar with marker non-terminals can be parsed by a given method, then the **SDT** can be implemented during parsing.

Postfix Translation Schemes

- In an **SDD** implementation, we can parse the grammar bottom-up and the **SDD** is **S-attributed**.
- An **SDT** is constructed such that the actions to be executed are placed at the end of the production and are executed only when the **RHS** of the production is reduced to **LHS** of the production i.e., reduction of the body to the head of the production.
- The **SDT**'s with all actions at the right end of the production bodies are called ***postfix SDT's*** or ***postfix syntax-directed translations***.

Postfix Translation Schemes

Example:

Obtain postfix SDT implementation of the desk calculator to evaluate the given expression.

Solution: **SDT** can be easily obtained by looking at the **SDD** shown below:

Productions	Semantic Rules
$S \rightarrow En$	S.val = 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
$F \rightarrow digit$	F.val = digit.lexval

Postfix Translation Schemes

Example: *Solution*

The ***Postfix SDT*** implementation of the desk calculator is shown below:

Productions	Actions
$S \rightarrow E$ 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 x F.val;}
$T \rightarrow F$	{T.val = F.val;}
$F \rightarrow (E)$	{F.val = E.val;}
$F \rightarrow$ digit	{F.val = digit.lexval;}

Parser-Stack Implementation of Postfix SDT's

- **`Postfix SDT's`** can be implemented during **`LR parsing`** by executing the actions when reductions occur.
- The attribute(s) of each grammar symbol can be put on the stack in a place where they can be found during the reduction.
- The best plan is to place the attributes along with the grammar symbols (or the **`LR`** states that represent these symbols) in records on the stack itself.

Parser-Stack Implementation of Postfix SDT's

In the following *Fig.* , the parser stack contains records with a field for a grammar symbol (or parser state) and, below it, a field for an attribute. The three grammar symbols **XYZ** are on top of the stack; perhaps they are about to be reduced according to a production like **A → XYZ**. Here, we show **X.x** as the one attribute of **X**, and so on.

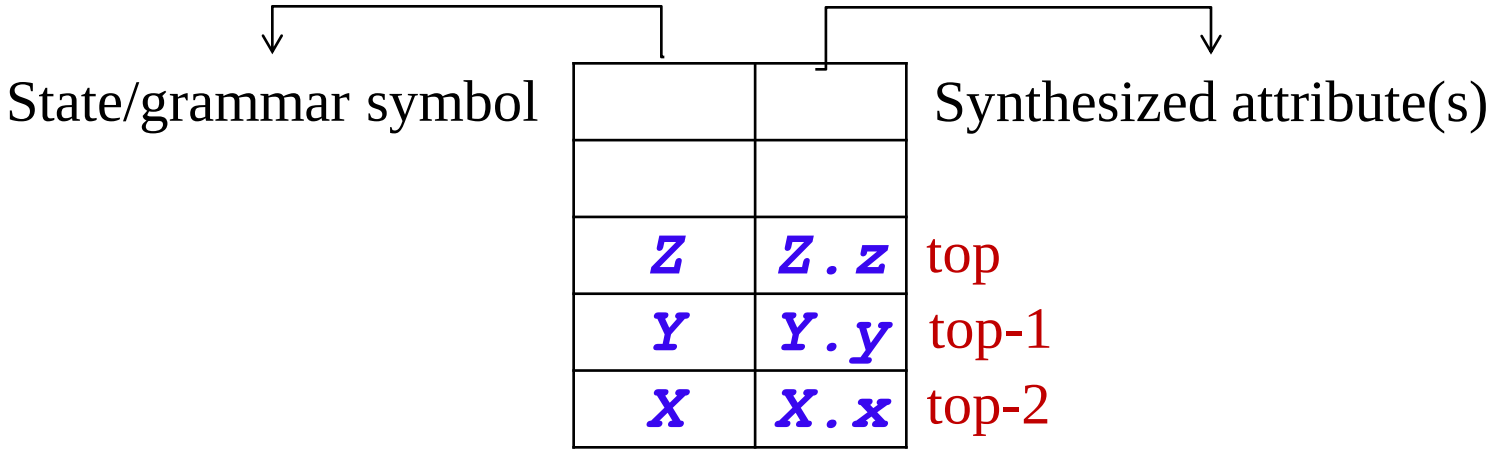

Fig: Parser stack with a field for synthesized attributes

Parser-Stack Implementation of Postfix SDT's

- If the attributes are all synthesized, and the actions occur at the ends of the productions, then we can compute the attributes for the head when we reduce the body to the head.
- If we reduce by a production such as ***A → XYZ***, then we have all the attributes of ***X***, ***Y*** , and ***Z*** available, at known positions on the stack, as shown in the above *Fig.*
- After the action, ***A*** and its attributes are at the top of the stack, in the position of the record for ***X***.

Parser-Stack Implementation of Postfix SDT's

Example: Implementing the *desk calculator* on a bottom-up parsing stack

Productions	Actions
$S \rightarrow E$ n	{print(*stack*[*top*-1].*val*); *top* = *top*-1;}
$E \rightarrow E_1$ + T	{*stack*[*top*-2].*val* = *stack*[*top*-2].*val* + *stack*[*top*].*val*; *top* = *top*-2;}
$E \rightarrow T$	
$T \rightarrow T_1$ * F	{*stack*[*top*-2].*val* = *stack*[*top*-2].*val* X *stack*[*top*].*val*; *top* = *top*-2;}
$T \rightarrow F$	
$F \rightarrow$ (E)	{*stack*[*top*-2].*val* = *stack*[*top*-1].*val*; *top* = *top*-2;}
$F \rightarrow$ digit	

Parser-Stack Implementation of Postfix SDT's

Example:

- Suppose that the stack is kept in an array of records called stack, with top a cursor to the top of the stack.
- Thus, `stack[top]` refers to the top record on the stack, `stack[top - 1]` to the record below that, and so on.
- Also, we assume that each record has a field called `val`, which holds the attribute of whatever grammar symbol is represented in that record.
- Thus, we may refer to the attribute `E.val` that appears at the third position on the stack as `stack[top - 2].val`.

Parser-Stack Implementation of Postfix SDT's

Example:

- For instance, in the second production, $E \rightarrow E_1 + T$, we go two positions below the top to get the value of E_1, and we find the value of T at the top. The resulting sum is placed where the head E will appear after the reduction, that is, two positions below the current top. The reason is that after the reduction, the three topmost stack symbols are replaced by one. After computing $E.val$, we pop two symbols off the top of the stack, so the record where we placed $E.val$ will now be at the top of the stack.

Parser-Stack Implementation of Postfix SDT's

Example:

- In the third production, **E → T**, no action is necessary, because the length of the stack does not change, and the value of **T.val** at the stack top will simply become the value of **E.val**.
- The same observation applies to the productions **T → F** and **T → digit**.
- Production **F → (E)** is slightly different. Although the value does not change, two positions are removed from the stack during the reduction, so the value has to move to the position after the reduction.

SDT's With Actions Inside Productions

- An action may be placed at any position within the body of a production.
- It is performed immediately after all symbols to its left are processed.
- Thus, if we have a production **`B → X {a} Y`** , the action a is done after we have recognized **`X`** (if **`X`** is a terminal) or all the terminals derived from **`X`** (if **`X`** is a nonterminal).

More precisely,

- If the parse is bottom-up, then we perform action **`a`** as soon as this occurrence of **`X`** appears on the top of the parsing stack.
- If the parse is top-down, we perform **`a`** just before we attempt to expand this occurrence of **`Y`** (if **`Y`** a nonterminal) or check for **`Y`** on the input (if **`Y`** is a terminal).

SDT's With Actions Inside Productions

Example: Problematic SDT for infix-to-prefix translation during parsing

As an extreme example of a problematic SDT, suppose that we turn our desk-calculator running example into an SDT that prints the prefix form of an expression, rather than evaluating the expression. The productions and actions are shown below:

1) $S \rightarrow E$ n
2) $E \rightarrow$ {print('+');} E_1 + T
3) $E \rightarrow T$
4) $T \rightarrow$ {print('*');} T_1 * F
5) $T \rightarrow F$
6) $F \rightarrow$ (E)
7) $F \rightarrow$ digit {print(digit.*lexval*);}

SDT's With Actions Inside Productions

Example: Problematic SDT for infix-to-prefix translation during parsing

- Unfortunately, it is impossible to implement this SDT during either top-down or bottom-up parsing, because the parser would have to perform critical actions, like printing instances of `*` or `+`, long before it knows whether these symbols will appear in its input.
- Using marker nonterminals M_2 and M_4 for the actions in productions `2` and `4`, respectively, on input that is a digit, a shift-reduce parser has conflicts between reducing by $M_2 \rightarrow \epsilon$, reducing by $M_4 \rightarrow \epsilon$, and shifting the digit.

SDT's With Actions Inside Productions

Any SDT can be implemented as follows:

1. Ignoring the actions, parse the input and produce a parse tree as a result.
2. Then, examine each interior node **N**, say one for production **A** → **α**. Add additional children to **N** for the actions in **α**, so the children of **N** from left to right have exactly the symbols and actions of **α**.
3. Perform a preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action.

SDT's With Actions Inside Productions

For instance, the following *Fig.* shows the parse tree for expression `3 * 5 + 4` with actions inserted. If we visit the nodes in preorder, we get the prefix form of the expression: `+ * 3 5 4`.

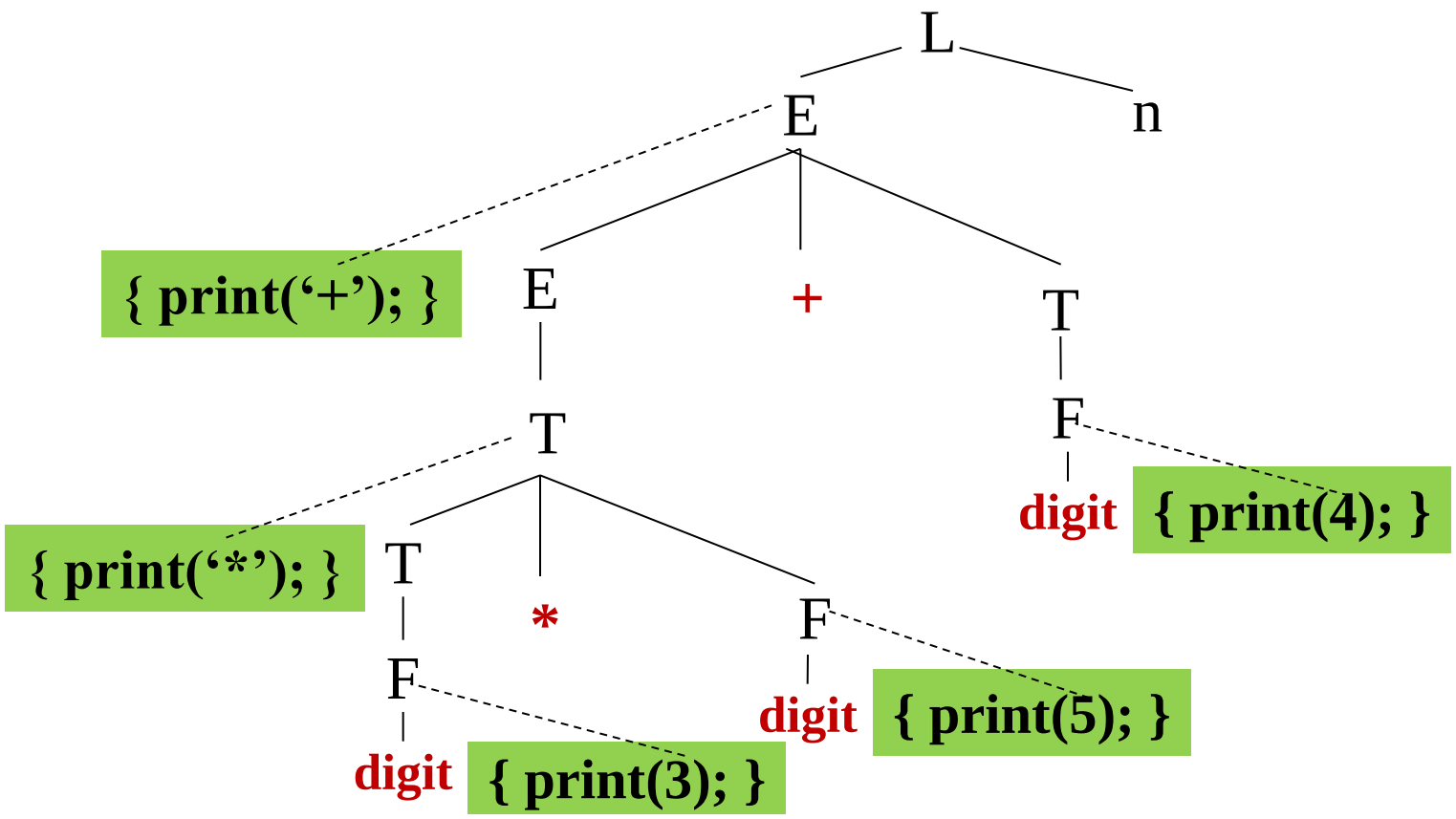

Fig: Parse tree with actions embedded

Summary...

Syntax-Directed Translation (SDT) Schemes

- Postfix Translation Schemes
- Parser-Stack Implementation of Postfix SDT's
- SDT's With Actions Inside Productions

Reading: Aho2, Section 5.4.1 to 5.4.3