Syntax Directed Translation

Outline

- Syntax Directed Definitions
- Evaluation Orders of SDD's
- Applications of Syntax Directed Translation
- Syntax Directed Translation Schemes

Introduction

- We can associate information with a language construct by attaching attributes to the grammar symbols.

- A syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.

Production	Semantic Rule
E->E1+T	E.code=E1.code||T.code||'+'

- We may alternatively insert the semantic actions inside the grammar

E -> E1+T {print '+'}

Syntax Directed Definitions

- A SDD is a context free grammar with attributes and rules
- Attributes are associated with grammar symbols and rules with productions
- Attributes may be of many kinds: numbers, types, table references, strings, etc.
- Synthesized attributes
  - A synthesized attribute at node N is defined only in terms of attribute values of children of N and at N it
- Inherited attributes
  - An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings

Example of S-attributed SDD

Production   Semantic Rules

1) L -> E n   L.val = E.val
2) E -> E1 + T   E.val = E1.val + T.val
3) E -> T   E.val = T.val
4) T -> T1 * F   T.val = T1.val * F.val
5) T -> F   T.val = F.val
6) F -> (E)   F.val = E.val
7) F -> digit   F.val = digit.lexval

Example of mixed attributes

Production

1) T -> FT'

2) T' -> *FT'$_1$

3) T' -> ε
1) F -> digit

Semantic Rules

T'.inh = F.val
T.val = T'.syn
T'$_1$.inh = T'.inh*F.val
T'.syn = T'$_1$.syn
T'.syn = T'.inh
F.val = F.val =
        digit.lexval

Evaluation orders for SDD's

- A dependency graph is used to determine the order of computation of attributes

- Dependency graph

  - For each parse tree node, the parse tree has a node for each attribute associated with that node

  - If a semantic rule defines the value of synthesized attribute A.b in terms of the value of X.c then the dependency graph has an edge from X.c to A.b

  - If a semantic rule defines the value of inherited attribute B.c in terms of the value of X.a then the dependency graph has an edge from X.c to B.c

- Example!

Ordering the evaluation of attributes

- If dependency graph has an edge from M to N then M must be evaluated before the attribute of N
- Thus the only allowable orders of evaluation are those sequence of nodes N1,N2,…,Nk such that if there is an edge from Ni to Nj then i<j
- Such an ordering is called a topological sortof a graph
- Example!

S-Attributed definitions

- An SDD is S-attributed if every attribute is synthesized
- We can have a post-order traversal of parse-tree to evaluate attributes in S-attributed definitions

```
postorder(N) {
        for (each child C of N, from the left) postorder(C);
        evaluate the attributes associated with node N;
    }
```

- S-Attributed definitions can be implemented during bottom-up parsing without the need to explicitly create parse trees

L-Attributed definitions

- A SDD is L-Attributed if the edges in dependency graph goes from Left to Right but not from Right to Left.
- More precisely, each attribute must be either
  - Synthesized
  - Inherited, but if there us a production A->X1X2…Xn and there is an inherited attribute Xi.a computed by a rule associated with this production, then the rule may only use:
    - Inherited attributes associated with the head A
    - Either inherited or synthesized attributes associated with the occurrences of symbols X1,X2,…,Xi-1 located to the left of Xi
    - Inherited or synthesized attributes associated with this occurrence of Xi itself, but in such a way that there is no cycle in the graph

Application of Syntax Directed Translation

- Type checking and intermediate code generation (chapter 6)
- Construction of syntax trees
  - Leaf nodes: Leaf(op,val)
  - Interior node: Node(op,c1,c2,…,ck)
- Example:

Production	Semantic Rules
1) E -> E1 + T	E.node=new node('+', E1.node,T.node)
2) E -> E1 - T	E.node=new node('-', E1.node,T.node)
3) E -> T	E.node = T.node
4) T -> (E)	T.node = E.node
5) T -> id	T.node = new Leaf(id,id.entry)
6) T -> num	T.node = new Leaf(num,num.val)

Syntax tree for L-attributed definition

Production

1) E -> TE'

2) E' -> + TE1'

3) E' -> -TE1'

4) E' -> ∈

5) T -> (E)

6) T -> id
7) T -> num

Semantic Rules

E.node=E'.syn   +
E'.inh=T.node
E1'.inh=new node('+', E'.inh,T.node)
E'.syn=E1'.syn
E1'.inh=new node('+', E'.inh,T.node)
E'.syn=E1'.syn
E'.syn = E'.inh

T.node = E.node

T.node=new Leaf(id,id.entry)
T.node = new Leaf(num,num.val)

Syntax directed translation schemes

- An SDT is a Context Free grammar with program fragments embedded within production bodies
- Those program fragments are called semantic actions
- They can appear at any position within production body
- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth first order
- Typically SDT's are implemented during parsing without building a parse tree

Postfix translation schemes

- Simplest SDDs are those that we can parse the grammar bottom-up and the SDD is s-attributed

- For such cases we can construct SDT where each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production

- SDT's with all actions at the right ends of the production bodies are called postfix SDT's

Example of postfix SDT

1) L -> E n   {print(E.val);}
2) E -> E1 + T   {E.val=E1.val+T.val;}
3) E -> T   {E.val = T.val;}
4) T -> T1 * F   {T.val=T1.val*F.val;}
5) T -> F   {T.val=F.val;}
6) F -> (E)   {F.val=E.val;}
7) F -> digit   {F.val=digit.lexval;}

Parse-Stack implementation of postfix SDT's

- In a shift-reduce parser we can easily implement semantic action using the parser stack

- For each nonterminal (or state) on the stack we can associate a record holding its attributes

- Then in a reduction step we can execute the semantic action at the end of a production to evaluate the attribute(s) of the non-terminal at the leftside of the production

- And put the value on the stack in replace of the rightside of production

Example

L -> E n   {print(stack[top-1].val);
                       top=top-1;}
E -> E1 + T   {stack[top-2].val=stack[top-2].val+stack.val;
                       top=top-2;}
E -> T
T -> T1 * F   {stack[top-2].val=stack[top-2].val+stack.val;
                       top=top-2;}
T -> F
F -> (E)   {stack[top-2].val=stack[top-1].val
   top=top-2;}
F -> digit

SDT's with actions inside productions

- For a production B->X {a} Y
  - If the parse is bottom-up then we perform action "a" as soon as this occurrence of X appears on the top of the parser stack
  - If the parser is top down we perform "a" just before we expand Y
- Sometimes we cant do things as easily as explained above
- One example is when we are parsing this SDT with a bottom-up parser

1) L -> E n
2) E -> {print('+');} E1 + T
3) E -> T
4) T -> {print('*');} T1 * F
5) T -> F
6) F -> (E)
7) F -> digit {print(digit.lexval);}

SDT's with actions inside productions (cont)

- Any SDT can be implemented as follows
  1. Ignore the actions and produce a parse tree
  2. Examine each interior node N and add actions as new children at the correct position
  3. Perform a postorder traversal and execute actions when their nodes are visited

L
E
{print('+');}   E   +   T
T   F
{print('*');}   T   *   F   {print(4);}
F   digit   {print(5);}
{print(3);}   digit
digit
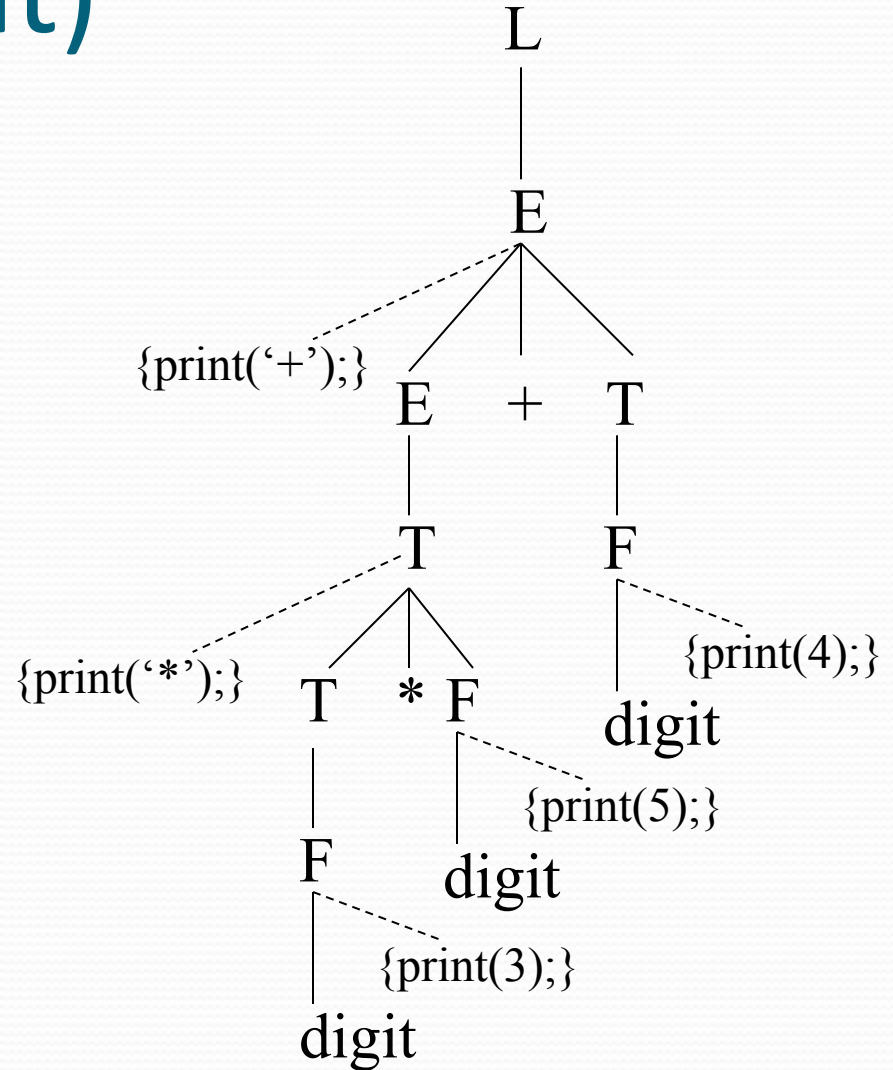
SDT's for L-Attributed definitions

- We can convert an L-attributed SDD into an SDT using following two rules:
  - Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A. if several inherited attributes of A are dpendent on one another in an acyclic fashion, order them so that those needed first are computed first
  - Place the action of a synthesized attribute for the head of a production at the end of the body of the production

Example

S -> while (C) S1 L1=new();

                 L2=new();

                 S1.next=L1;

                 C.false=S.next;

                 C.true=L2;

                 S.code=label||L1||C.code||label||L2||S1.code

S -> while ({L1=new();L2=new();C.false=S.next;C.true=L2;}
C) {S1.next=L1;}
S1{S.code=label||L1||C.code||label||L2||S1.code;}