Syntax-Directed Translation

Lecture 8

Objectives

By the end of this lecture you should be able to:

1. Identify the difference between SDDs and SDTs.
2. Find the side effects of parsing with an SDD or an SDT.
3. Determine whether an attribute of an SDD is synthesized or inherited.
4. Construct SDDs and SDTs.
5. Eliminate left-recursion from an SDD with synthesized attributes.
6. Construct an SDT which is equivalent to a given SDD.

Outline

Outline

Semantic Analysis

- The semantic analysis of a source program amounts to *understanding* what the program means.
- Understanding what a program means is best demonstrated by translating the program to an intermediate representation which can be pretty directly mapped to equivalent machine language code.
- All aspects of a source program which are not accounted for by parsing are considered to fall under the rubric of semantic analysis; for example, type checking.
- We consider different translation schemes, all taking the result of parsing as a starting point.

Semantic Analysis

- The semantic analysis of a source program amounts to *understanding* what the program means.
- Understanding what a program means is best demonstrated by translating the program to an intermediate representation which can be pretty directly mapped to equivalent machine language code.
- All aspects of a source program which are not accounted for by parsing are considered to fall under the rubric of semantic analysis; for example, type checking.
- We consider different translation schemes, all taking the result of parsing as a starting point.

Semantic Analysis

- The semantic analysis of a source program amounts to *understanding* what the program means.
- Understanding what a program means is best demonstrated by translating the program to an intermediate representation which can be pretty directly mapped to equivalent machine language code.
- All aspects of a source program which are not accounted for by parsing are considered to fall under the rubric of semantic analysis; for example, type checking.
- We consider different translation schemes, all taking the result of parsing as a starting point.

Semantic Analysis

- The semantic analysis of a source program amounts to *understanding* what the program means.
- Understanding what a program means is best demonstrated by translating the program to an intermediate representation which can be pretty directly mapped to equivalent machine language code.
- All aspects of a source program which are not accounted for by parsing are considered to fall under the rubric of semantic analysis; for example, type checking.
- We consider different translation schemes, all taking the result of parsing as a starting point.

Syntax-Directed Translation

Syntax-Directed Definitions (SDDs):

- Symbols of the grammar have associated attributes.
- Values of attributes are computed by augmenting grammar rules with actions.
- Grammar symbols have distinguished attributes which carry the "meaning" (translation) of instances of the symbol.
- Better suited for specification.

Syntax-Directed Translation Schemes (SDTs):

- Pieces of code to perform translation actions are added at various positions in the right-side of a production.
- The actions directly generate the translation.
- Better suited for implementation.

Syntax-Directed Translation

Syntax-Directed Definitions (SDDs):

- Symbols of the grammar have associated attributes.
- Values of attributes are computed by augmenting grammar rules with actions.
- Grammar symbols have distinguished attributes which carry the "meaning" (translation) of instances of the symbol.
- Better suited for specification.

Syntax-Directed Translation Schemes (SDTs):

- Pieces of code to perform translation actions are added at various positions in the right-side of a production.
- The actions directly generate the translation.
- Better suited for implementation.

Syntax-Directed Translation

Syntax-Directed Definitions (SDDs):

- Symbols of the grammar have associated attributes.
- Values of attributes are computed by augmenting grammar rules with actions.
- Grammar symbols have distinguished attributes which carry the "meaning" (translation) of instances of the symbol.
- Better suited for specification.

Syntax-Directed Translation Schemes (SDTs):

- Pieces of code to perform translation actions are added at various positions in the right-side of a production.
- The actions directly generate the translation.
- Better suited for implementation.

Syntax-Directed Translation

Syntax-Directed Definitions (SDDs):

- Symbols of the grammar have associated attributes.
- Values of attributes are computed by augmenting grammar rules with actions.
- Grammar symbols have distinguished attributes which carry the “meaning” (translation) of instances of the symbol.
- Better suited for specification.

Syntax-Directed Translation Schemes (SDTs):

- Pieces of code to perform translation actions are added at various positions in the right-side of a production.
- The actions directly generate the translation.
- Better suited for implementation.

Syntax-Directed Translation

Syntax-Directed Definitions (SDDs):

- Symbols of the grammar have associated attributes.
- Values of attributes are computed by augmenting grammar rules with actions.
- Grammar symbols have distinguished attributes which carry the “meaning” (translation) of instances of the symbol.
- Better suited for specification.

Syntax-Directed Translation Schemes (SDTs):

- Pieces of code to perform translation actions are added at various positions in the right-side of a production.
- The actions directly generate the translation.
- Better suited for implementation.

Syntax-Directed Translation

Syntax-Directed Definitions (SDDs):

- Symbols of the grammar have associated attributes.
- Values of attributes are computed by augmenting grammar rules with actions.
- Grammar symbols have distinguished attributes which carry the “meaning” (translation) of instances of the symbol.
- Better suited for specification.

Syntax-Directed Translation Schemes (SDTs):

- Pieces of code to perform translation actions are added at various positions in the right-side of a production.
- The actions directly generate the translation.
- Better suited for implementation.

Syntax-Directed Translation

Syntax-Directed Definitions (SDDs):

- Symbols of the grammar have associated attributes.
- Values of attributes are computed by augmenting grammar rules with actions.
- Grammar symbols have distinguished attributes which carry the "meaning" (translation) of instances of the symbol.
- Better suited for specification.

Syntax-Directed Translation Schemes (SDTs):

- Pieces of code to perform translation actions are added at various positions in the right-side of a production.
- The actions directly generate the translation.
- Better suited for implementation.

Syntax-Directed Translation

Syntax-Directed Definitions (SDDs):

- Symbols of the grammar have associated attributes.
- Values of attributes are computed by augmenting grammar rules with actions.
- Grammar symbols have distinguished attributes which carry the “meaning” (translation) of instances of the symbol.
- Better suited for specification.

Syntax-Directed Translation Schemes (SDTs):

- Pieces of code to perform translation actions are added at various positions in the right-side of a production.
- The actions directly generate the translation.
- Better suited for implementation.

Syntax-Directed Translation

Syntax-Directed Definitions (SDDs):

- Symbols of the grammar have associated attributes.
- Values of attributes are computed by augmenting grammar rules with actions.
- Grammar symbols have distinguished attributes which carry the "meaning" (translation) of instances of the symbol.
- Better suited for specification.

Syntax-Directed Translation Schemes (SDTs):

- Pieces of code to perform translation actions are added at various positions in the right-side of a production.
- The actions directly generate the translation.
- Better suited for implementation.

A Note on Notation

We will often need to distinguish different occurrences of the same grammar symbol in grammar productions. Subscripts will be used to do so. Thus, in

$$E \longrightarrow E_1 + T$$

E_1 is not a symbol different from E; it is a different occurrence of E.

Infix-to-Prefix

Example (SDD)

$$
\begin{array}{llll}
E & \longrightarrow & E_1 + T & E.code = {'+'} \circ E_1.code \circ T.code \\
E & \longrightarrow & T & E.code = T.code \\
T & \longrightarrow & T_1 * F & T.code = {'*'} \circ T_1.code \circ F.code \\
T & \longrightarrow & F & T.code = F.code \\
F & \longrightarrow & (E) & F.code = E.code \\
F & \longrightarrow & \mathbf{id} & F.code = \mathbf{id}.lexval
\end{array}
$$

Infix-to-Prefix

Example (SDT)

$$
\begin{array}{lcl}
E & \longrightarrow & \{\texttt{print}('+')\}E_1 + T \\
E & \longrightarrow & T \\
T & \longrightarrow & \{\texttt{print}('*')\}T_1 * F \\
T & \longrightarrow & F \\
F & \longrightarrow & (E) \\
F & \longrightarrow & \mathbf{id}\ \{\texttt{print}(\mathbf{id}.lexval)\}
\end{array}
$$

Outline

Synthesized Attributes

Definition

1. An attribute a of an occurrence A_i of a variable A is a synthesized attribute if, $A_i \rightarrow \alpha$ is a rule where the value of a is determined only by the values of other attributes of A_i and the values of attributes of symbols in α.
2. All attributes of terminals are synthesized attributes; their values are provided by the lexical analyzer.

Example (Infix-to-Prefix SDD)

The attribute *code* is a synthesized attribute of E, T, and F in the infix-to-prefix SDD. The attribute *lexval* is a synthesized attribute of **id**.

Synthesized Attributes

Definition

1. An attribute a of an occurrence A_i of a variable A is a synthesized attribute if, $A_i \rightarrow \alpha$ is a rule where the value of a is determined only by the values of other attributes of A_i and the values of attributes of symbols in α.
2. All attributes of terminals are synthesized attributes; their values are provided by the lexical analyzer.

Example (Infix-to-Prefix SDD)

The attribute *code* is a synthesized attribute of E, T, and F in the infix-to-prefix SDD. The attribute *lexval* is a synthesized attribute of **id**.

Inherited Attributes

Definition

An attribute a of an occurrence A_i of a variable A is an inherited attribute if, $B \rightarrow \alpha A_i \beta$ is a rule where the value of a is determined by the values of attributes of symbols occurring in the rule.

Example

$$B \longrightarrow B_1 A C \quad A.inh = B.inh \circ B_1.syn \circ A.syn \circ C.syn$$

Inherited Attributes

Definition

An attribute a of an occurrence A_i of a variable A is an inherited attribute if, $B \rightarrow \alpha A_i \beta$ is a rule where the value of a is determined by the values of attributes of symbols occurring in the rule.

Example

$$B \longrightarrow B_1 A C \quad A.inh = B.inh \circ B_1.syn \circ A.syn \circ C.syn$$

SDDs and Parse Trees

- It is convenient to suppose that parsing constructs a parse tree.
- SDD rules are applied to compute the values of attributes at various nodes of the tree.
- In general, all attribute values needed for a computation must be computed prior to this computation; this constrains the order in which we traverse the tree.
- If all attributes are synthesized, any bottom-up order will do—one resulting from a postorder traversal, for example.
- The parse tree with the values of attributes indicated is called an annotated parse tree.

SDDs and Parse Trees

- It is convenient to suppose that parsing constructs a parse tree.
- SDD rules are applied to compute the values of attributes at various nodes of the tree.
- In general, all attribute values needed for a computation must be computed prior to this computation; this constrains the order in which we traverse the tree.
- If all attributes are synthesized, any bottom-up order will do—one resulting from a postorder traversal, for example.
- The parse tree with the values of attributes indicated is called an annotated parse tree.

SDDs and Parse Trees

- It is convenient to suppose that parsing constructs a parse tree.
- SDD rules are applied to compute the values of attributes at various nodes of the tree.
- In general, all attribute values needed for a computation must be computed prior to this computation; this constrains the order in which we traverse the tree.
- If all attributes are synthesized, any bottom-up order will do—one resulting from a postorder traversal, for example.
- The parse tree with the values of attributes indicated is called an annotated parse tree.

SDDs and Parse Trees

- It is convenient to suppose that parsing constructs a parse tree.
- SDD rules are applied to compute the values of attributes at various nodes of the tree.
- In general, all attribute values needed for a computation must be computed prior to this computation; this constrains the order in which we traverse the tree.
- If all attributes are synthesized, any bottom-up order will do—one resulting from a postorder traversal, for example.
- The parse tree with the values of attributes indicated is called an annotated parse tree.

SDDs and Parse Trees

- It is convenient to suppose that parsing constructs a parse tree.
- SDD rules are applied to compute the values of attributes at various nodes of the tree.
- In general, all attribute values needed for a computation must be computed prior to this computation; this constrains the order in which we traverse the tree.
- If all attributes are synthesized, any bottom-up order will do—one resulting from a postorder traversal, for example.
- The parse tree with the values of attributes indicated is called an annotated parse tree.

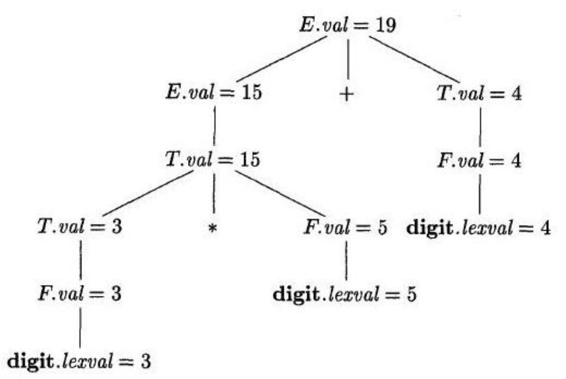
Expression SDD

Example (SDD)

$$
\begin{array}{llll}
E & \longrightarrow & E_1 + T & E.val = E_1.val + T.val \\
E & \longrightarrow & T & E.val = T.val \\
T & \longrightarrow & T_1 * F & T.val = T_1.val \times F.val \\
T & \longrightarrow & F & T.val = F.val \\
F & \longrightarrow & (E) & F.val = E.val \\
F & \longrightarrow & \textbf{digit} & F.val = \textbf{digit}.lexval
\end{array}
$$

Expression SDD

Example (Annotated Parse Tree)



©Aho et al. (2007)

Why Inherited Attributes?

- Why would we ever need inherited attributes?
- Grammar transformations may produce awkward grammars for which synthesized attributes do not suffice.
- Typical example: eliminating left recursion.

Why Inherited Attributes?

- Why would we ever need inherited attributes?
- Grammar transformations may produce awkward grammars for which synthesized attributes do not suffice.
- Typical example: eliminating left recursion.

Why Inherited Attributes?

- Why would we ever need inherited attributes?
- Grammar transformations may produce awkward grammars for which synthesized attributes do not suffice.
- Typical example: eliminating left recursion.

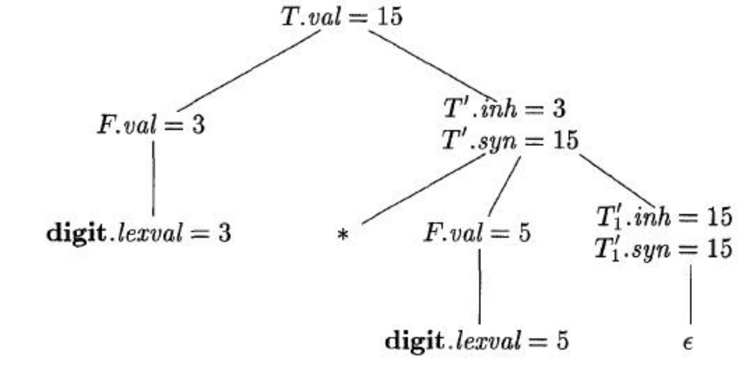
Expression SDD Without Left Recursion

Example (SDD)

$$
\begin{array}{llll}
T & \longrightarrow & FT' & T'.inh = F.val \\
 & & & T.val = T'.syn \\
T' & \longrightarrow & *FT'_1 & T'_1.inh = T'.inh \times F.val \\
 & & & T'.syn = T'_1.syn \\
T' & \longrightarrow & \varepsilon & T'.syn = T'.inh \\
F & \longrightarrow & \textbf{digit} & F.val = \textbf{digit}.lexval
\end{array}
$$

Expression SDD Without Left Recursion

Example (Tree)



ⒸAho et al. (2007)

Outline

Left Recursion in SDDs

- Suppose we are given an SDD with a left recursive grammar having only synthesized attributes.
- We have seen how to construct an equivalent grammar which is not left-recursive.
- But how do we get an equivalent SDD?
- We shall see how to do so if we only have immediate left recursion.
- Can you extend it to the general case?

Left Recursion in SDDs

- Suppose we are given an SDD with a left recursive grammar having only synthesized attributes.
- We have seen how to construct an equivalent grammar which is not left-recursive.
- But how do we get an equivalent SDD?
- We shall see how to do so if we only have immediate left recursion.
- Can you extend it to the general case?

Left Recursion in SDDs

- Suppose we are given an SDD with a left recursive grammar having only synthesized attributes.
- We have seen how to construct an equivalent grammar which is not left-recursive.
- But how do we get an equivalent SDD?
- We shall see how to do so if we only have immediate left recursion.
- Can you extend it to the general case?

Left Recursion in SDDs

- Suppose we are given an SDD with a left recursive grammar having only synthesized attributes.
- We have seen how to construct an equivalent grammar which is not left-recursive.
- But how do we get an equivalent SDD?
- We shall see how to do so if we only have immediate left recursion.
- Can you extend it to the general case?

Left Recursion in SDDs

- Suppose we are given an SDD with a left recursive grammar having only synthesized attributes.
- We have seen how to construct an equivalent grammar which is not left-recursive.
- But how do we get an equivalent SDD?
- We shall see how to do so if we only have immediate left recursion.
- Can you extend it to the general case?

Eliminating Left Recursion

We are given

$$\begin{array}{lcll} A & \longrightarrow & A_1 Y & A.a = g(A_1.a, Y.y) \\ A & \longrightarrow & X & A.a = f(X.x) \end{array}$$

We get

$$\begin{array}{lcll} A & \longrightarrow & XR & R.i = f(X.x) \\ & & & A.a = R.s \\ R & \longrightarrow & YR_1 & R_1.i = g(R.i, Y.y) \\ & & & R.s = R_1.s \\ R & \longrightarrow & \varepsilon & R.s = R.i \end{array}$$

Eliminating Left Recursion

We are given

$$
\begin{array}{llll}
A & \longrightarrow & A_1 Y & A.a = g(A_1.a, Y.y) \\
A & \longrightarrow & X & A.a = f(X.x)
\end{array}
$$

We get

$$
\begin{array}{llll}
A & \longrightarrow & XR & R.i = f(X.x) \\
 & & & A.a = R.s \\
R & \longrightarrow & YR_1 & R_1.i = g(R.i, Y.y) \\
 & & & R.s = R_1.s \\
R & \longrightarrow & \varepsilon & R.s = R.i
\end{array}
$$

Outline

The Problem

- We are given an SDD and a parse tree for a given string.
- We would like to apply the rules of the SDD.
- We want to make sure that attribute values needed for a computation are available prior to the computation.
- How do we do so?

The Problem

- We are given an SDD and a parse tree for a given string.
- We would like to apply the rules of the SDD.
- We want to make sure that attribute values needed for a computation are available prior to the computation.
- How do we do so?

The Problem

- We are given an SDD and a parse tree for a given string.
- We would like to apply the rules of the SDD.
- We want to make sure that attribute values needed for a computation are available prior to the computation.
- How do we do so?

The Problem

- We are given an SDD and a parse tree for a given string.
- We would like to apply the rules of the SDD.
- We want to make sure that attribute values needed for a computation are available prior to the computation.
- How do we do so?

Dependency Graphs

- We assume that parse tree nodes are labelled by subscripted grammar symbols.

Definition

Let $T = (N_T, E_T)$ be a parse tree and let D be the associated SDD. The dependency graph of T and D is a graph $G_D(T) = (N_G, E_G)$ where

Dependency Graphs

- We assume that parse tree nodes are labelled by subscripted grammar symbols.

Definition

Let $T = (N_T, E_T)$ be a parse tree and let D be the associated SDD. The dependency graph of T and D is a graph $G_D(T) = (N_G, E_G)$ where

- $N_G = \{n.a \mid n \in N_T$ and a is an attribute of $n\}$
- $(u.a, v.b) \in E_G$ if and only if

Dependency Graphs

- We assume that parse tree nodes are labelled by subscripted grammar symbols.

Definition

Let $T = (N_T, E_T)$ be a parse tree and let D be the associated SDD. The dependency graph of T and D is a graph $G_D(T) = (N_G, E_G)$ where

- $N_G = \{n.a | n \in N_T$ and a is an attribute of $n\}$
- $(u.a, v.b) \in E_G$ if and only if
  1. b is a synthesized attribute of v, u is a child of v in T, and the corresponding production has a rule which defines $v.b$ in terms of $u.a$.
  2. b is an inherited attribute of b, $u = v$ or u is a sibling or parent of v in T, and the corresponding production has a rule which defines $v.b$ in terms of $u.a$.

Dependency Graphs

- We assume that parse tree nodes are labelled by subscripted grammar symbols.

Definition

Let $T = (N_T, E_T)$ be a parse tree and let D be the associated SDD. The dependency graph of T and D is a graph $G_D(T) = (N_G, E_G)$ where

- $N_G = \{n.a | n \in N_T$ and a is an attribute of $n\}$
- $(u.a, v.b) \in E_G$ if and only if
  1. b is a synthesized attribute of v, u is a child of v in T, and the corresponding production has a rule which defines $v.b$ in terms of $u.a$.
  2. b is an inherited attribute of b, $u = v$ or u is a sibling or parent of v in T, and the corresponding production has a rule which defines $v.b$ in terms of $u.a$.

Dependency Graphs

- We assume that parse tree nodes are labelled by subscripted grammar symbols.

Definition

Let $T = (N_T, E_T)$ be a parse tree and let D be the associated SDD. The dependency graph of T and D is a graph $G_D(T) = (N_G, E_G)$ where

- $N_G = \{n.a | n \in N_T$ and a is an attribute of $n\}$
- $(u.a, v.b) \in E_G$ if and only if
  1. b is a synthesized attribute of v, u is a child of v in T, and the corresponding production has a rule which defines $v.b$ in terms of $u.a$.
  2. b is an inherited attribute of b, $u = v$ or u is a sibling or parent of v in T, and the corresponding production has a rule which defines $v.b$ in terms of $u.a$.

Dependency Graphs

- We assume that parse tree nodes are labelled by subscripted grammar symbols.

Definition

Let $T = (N_T, E_T)$ be a parse tree and let D be the associated SDD. The dependency graph of T and D is a graph $G_D(T) = (N_G, E_G)$ where

- $N_G = \{n.a | n \in N_T$ and a is an attribute of $n\}$
- $(u.a, v.b) \in E_G$ if and only if
  1. b is a synthesized attribute of v, u is a child of v in T, and the corresponding production has a rule which defines $v.b$ in terms of $u.a$.
  2. b is an inherited attribute of b, $u = v$ or u is a sibling or parent of v in T, and the corresponding production has a rule which defines $v.b$ in terms of $u.a$.

Dependency Graphs

- We assume that parse tree nodes are labelled by subscripted grammar symbols.

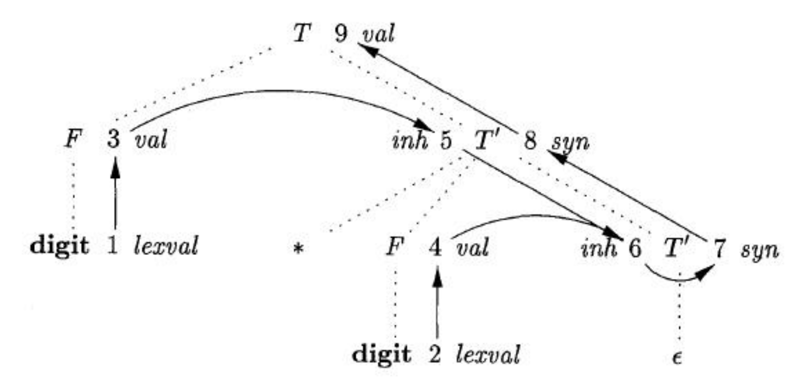
Definition

Let $T = (N_T, E_T)$ be a parse tree and let D be the associated SDD. The dependency graph of T and D is a graph $G_D(T) = (N_G, E_G)$ where

- $N_G = \{n.a | n \in N_T$ and a is an attribute of $n\}$
- $(u.a, v.b) \in E_G$ if and only if
  1. b is a synthesized attribute of v, u is a child of v in T, and the corresponding production has a rule which defines $v.b$ in terms of $u.a$.
  2. b is an inherited attribute of b, $u = v$ or u is a sibling or parent of v in T, and the corresponding production has a rule which defines $v.b$ in terms of $u.a$.

Expression SDD Without Left Recursion

Example (Dependency Graph)



©Aho et al. (2007)

Order of Evaluation

- If the dependency graph is a directed acyclic graph (DAG), a topological sort of its nodes yields a valid order of evaluation:
  1. Initialize an empty queue.
  2. While there are nodes in the graph.
     1. Find a node n with zero in-degree and enqueue it.
     2. Remove n and all arcs emanating from it from the graph.
  3. Evaluate a node once it is dequeued.

Order of Evaluation

- If the dependency graph is a directed acyclic graph (DAG), a topological sort of its nodes yields a valid order of evaluation:
  1. Initialize an empty queue.
  2. While there are nodes in the graph.
     1. Find a node n with zero in-degree and enqueue it.
     2. Remove n and all arcs emanating from it from the graph.
  3. Evaluate a node once it is dequeued.

Order of Evaluation

- If the dependency graph is a directed acyclic graph (DAG), a topological sort of its nodes yields a valid order of evaluation:
  1. Initialize an empty queue.
  2. While there are nodes in the graph.
     1. Find a node n with zero in-degree and enqueue it.
     2. Remove n and all arcs emanating from it from the graph.
  3. Evaluate a node once it is dequeued.

Order of Evaluation

- If the dependency graph is a directed acyclic graph (DAG), a topological sort of its nodes yields a valid order of evaluation:
  1. Initialize an empty queue.
  2. While there are nodes in the graph.
     1. Find a node n with zero in-degree and enqueue it.
     2. Remove n and all arcs emanating from it from the graph.
  3. Evaluate a node once it is dequeued.

Order of Evaluation

- If the dependency graph is a directed acyclic graph (DAG), a topological sort of its nodes yields a valid order of evaluation:
  1. Initialize an empty queue.
  2. While there are nodes in the graph.
     1. Find a node *n* with zero in-degree and enqueue it.
     2. Remove *n* and all arcs emanating from it from the graph.
  3. Evaluate a node once it is dequeued.

Cycles in the Dependency Graph

Example

$$
\begin{array}{lcll}
A & \longrightarrow & B & A.s = B.i \\
 & & & B.i = A.s + 1
\end{array}
$$

S-Attributed SDDs

Definition

An SDD is S-attributed if every attribute is synthesized.

Example

The left recursive expression grammar is an S-attributed grammar.

Observation

If D is an S-attributed SDD, then, for every parse tree T, $G_D(T)$ is a DAG.

S-Attributed SDDs

Definition

An SDD is S-attributed if every attribute is synthesized.

Example

The left recursive expression grammar is an S-attributed grammar.

Observation

If D is an S-attributed SDD, then, for every parse tree T, $G_D(T)$ is a DAG.

S-Attributed SDDs

Definition

An SDD is S-attributed if every attribute is synthesized.

Example

The left recursive expression grammar is an S-attributed grammar.

Observation

If D is an S-attributed SDD, then, for every parse tree T, $G_D(T)$ is a DAG.

L-Attributed SDDs

Definition

An SDD is *L-attributed* if for every rule $B \longrightarrow \alpha A \beta$

1. every inherited attribute a of A depends only on inherited attributes of B and attributes of symbols occurring in α; and
2. every synthesized attribute of B depends only on attributes of symbols in $\alpha A \beta$ and on inherited attributes of B.

Example

The non-left-recursive expression grammar is L-attributed.

Observation

If D is an L-attributed SDD, then, for every parse tree T, $G_D(T)$ is a DAG.

L-Attributed SDDs

Definition

An SDD is *L-attributed* if for every rule $B \longrightarrow \alpha A \beta$

1. every inherited attribute a of A depends only on inherited attributes of B and attributes of symbols occurring in α; and
2. every synthesized attribute of B depends only on attributes of symbols in $\alpha A \beta$ and on inherited attributes of B.

Example

The non-left-recursive expression grammar is L-attributed.

Observation

If D is an L-attributed SDD, then, for every parse tree T, $G_D(T)$ is a DAG.

L-Attributed SDDs

Definition

An SDD is L-attributed if for every rule $B \longrightarrow \alpha A \beta$

1. every inherited attribute a of A depends only on inherited attributes of B and attributes of symbols occurring in α; and
2. every synthesized attribute of B depends only on attributes of symbols in $\alpha A \beta$ and on inherited attributes of B.

Example

The non-left-recursive expression grammar is L-attributed.

Observation

If D is an L-attributed SDD, then, for every parse tree T, $G_D(T)$ is a DAG.

Outline

From SDDs to SDTs

Given an *L*-attributed SDD

1. Insert the action that computes an inherited attribute of a non-terminal immediately before the occurrence of the non-terminal on the right-side of a rule.
   - If several inherited attributes are computed, make sure the actions are ordered in a way that is consistent with the dependency graph.
2. Place the action which computes a synthesized attribute for a non-terminal in a rule at the end of that rule.

From SDDs to SDTs

Given an *L*-attributed SDD

1. Insert the action that computes an inherited attribute of a non-terminal immediately before the occurrence of the non-terminal on the right-side of a rule.
   - If several inherited attributes are computed, make sure the actions are ordered in a way that is consistent with the dependency graph.
2. Place the action which computes a synthesized attribute for a non-terminal in a rule at the end of that rule.

From SDDs to SDTs

Given an *L*-attributed SDD

1. Insert the action that computes an inherited attribute of a non-terminal immediately before the occurrence of the non-terminal on the right-side of a rule.
   - If several inherited attributes are computed, make sure the actions are ordered in a way that is consistent with the dependency graph.
2. Place the action which computes a synthesized attribute for a non-terminal in a rule at the end of that rule.

Expression SDT Without Left Recursion

Example

Following is an SDT corresponding to the earlier non-left-recursive SDD.

$$
\begin{array}{lcl}
T & \longrightarrow & F \ \{T'.inh = F.val\} \ T' \ \{T.val = T'.syn\} \\
T' & \longrightarrow & *F \ \{T'_1.inh = T'.inh \times F.val\} \ T'_1 \ \{T'.syn = T'_1.syn\} \\
T' & \longrightarrow & \varepsilon \ \{T'.syn = T'.inh\} \\
F & \longrightarrow & \textbf{digit} \ \{F.val = \textbf{digit}.lexval\}
\end{array}
$$