4. Semantic Analysis I

Syntax Directed Definitions – Syntax Directed Translation Schemes

Andrea Polini, Luca Tesei

Formal Languages and Compilers
MSc in Computer Science
University of Camerino

ToC

Where we are?

So far we were able to check:

- the program includes correct "words"
- "words" are combined in correct "sentences"

What's next?

- We would like to perform additional checks to increase guarantees of correctness
- We would like to transform the program from the source language into the target one, and according to precisely defined semantic rules

Where we are?

So far we were able to check:

- the program includes correct “words”
- “words” are combined in correct “sentences”

What’s next?

- We would like to perform additional checks to increase guarantees of correctness
- We would like to transform the program from the source language into the target one, and according to precisely defined semantic rules

Where we are?

So far we were able to check:

- the program includes correct “words”
- “words” are combined in correct “sentences”

What’s next?

- ▶ We would like to perform additional checks to increase guarantees of correctness
- ▶ We would like to transform the program from the source language into the target one, and according to precisely defined semantic rules

Additional checks

Additional Checks

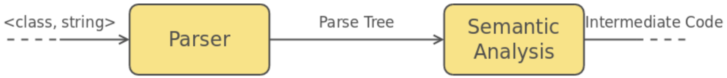
There are many additional checks that can be performed to increase correctness of code:

- Coherent usage of variables
  - definition-usage
  - type
- Existence of unreacheable code blocks
- ...

Semantic Analysis

In semantic analysis context sensitive analysis are performed without resurrecting to Context Sensitive grammar definitions. Here we focus on mechanisms for type checking and generation of intermediate code

Semantic analysis



ToC

Syntax Directed Definitions

Attributes

Attributes are used to associate characteristics and store values associated to grammar symbols.
A syntax directed definition provides the semantic rules to permit the definition of the values for the attributes

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \| \| T.code \| \|' +'$

- attributes are associated to grammar symbols and can be of any kind
- rules are associated to productions

Attributes

An SDD can be defined using two different kinds of attributes:

- Synthesized attributes: a synthesized attributes at node N is defined only in terms of attribute values at the children of N and at N itself
- 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

Attributes

Example

Consider the usual grammar and let's define a set of "reasonable" semantic rules:
$L \to E \quad E \to E + T \quad E \to T \quad T \to T * F \quad T \to F \quad F \to (E) \quad F \to id$

SDD and parse trees

An SDD with only synthesized attributes is called *S-attributed*

It is generally useful to represent attributes within parse trees. A parse tree showing the values of attributes is referred as an annotated parse tree

Order of evaluation for attributes

The order of evaluation of attributes should reflect the defined parsing strategy. In any case semantic rules impose an order of evalution that in case, inherited and synthetized attributes are present at the same time, is not guaranteed to exist.

Let's consider the expression "(3+4)*(5+6)" and let's derive its annotated parse tree from the semantic rules defined before

Inherited attributes example

Let's consider the non left recursive and factored grammar for expressions:
$E \rightarrow TE' \;\; E' \rightarrow +TE'|\epsilon \;\; T \rightarrow FT' \;\; T' \rightarrow *FT'|\epsilon \;\; F \rightarrow (E)|id$
define an SDD using as reference the parse tree for the sentence "$3 + 5 * 6$"

Evaluation Orders for SDD's

Dependency Graphs

A dependency graph represents the flow of information among the attribute instances in a particular parse tree.

- each attribute for a grammar symbol constitute a node in the graph
- syntesized attributes
- inherited attributes

Let's identify the dependency graph for the parse tree defined before, and let's compute the value of the various attributes

SDD with acyclic topological sort

S-attributed

If every attribute is synthesized the SDD is said S-attributed, in such a case an LR parser could even avoid the explicit derivation of the parse tree

L-attributed

Each attribute in the SDD satisfies one of the following conditions:

- it is synthesized
- it is inherited but it depends only from attributes on siblings on the left or inherited attributes associated to the parent symbol
- it is inherited or synthesized from attributes from the same symbol in a way that cycles are not generated

Semantic rules with controlled side effects

Side effects

A side effect consists of program fragment contained with semantic rules. It is necessary to control side effects is SDD in two possible ways:

- Permit incidental side effects
- Constraint admissible evaluation orders so to have the same translation with any admissible order.

Why to use them?

- to associate actions to carry on with specific steps of the compiler
- to print messages for the user useful during compilation
- to check correctness related aspects (e.g. types)

Semantic rules with controlled side effects

Side effects

A side effect consists of program fragment contained with semantic rules. It is necessary to control side effects is SDD in two possible ways:

- Permit incidental side effects
- Constraint admissible evaluation orders so to have the same translation with any admissible order.

Why to use them?

- to associate actions to carry on with specific steps of the compiler
- to print messages for the user useful during compilation
- to check correctness related aspects (e.g. types)

Semantic Rules with side effects

Example

Let's consider the following grammar:
$D \rightarrow TL; \quad T \rightarrow \textbf{int}|\textbf{float} \quad L \rightarrow L_1, \textbf{id}|\textbf{id}$
Let's add sematic rules to successively permit type checking

Semantic Rules with side effects

Exercise

Let's consider the following grammar that generates binary numbers with a decimal point:

$S \rightarrow L.L|L \quad L \rightarrow LB|B \quad B \rightarrow 0|1$

Design an L-attributed and an S-attributed SDD to make the translation in decimal numbers

Abstract Syntax Tree

Abstract Syntax Tree

Abstract Syntax Tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is “abstract” in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.
Syntax trees are useful for translation purpose making the phase much easier.

Let’s consider the sentence $(a + b) * 5$ over the grammar:

$E \rightarrow TE' \quad E' \rightarrow +TE'|\epsilon \quad T \rightarrow FT' \quad T' \rightarrow *FT'|\epsilon \quad F \rightarrow (E)|\textbf{id}|\textbf{num}$

Let’s build the parse tree and the AST

Using SDDs to build AST

To build a syntax tree two different kind of nodes need to be created, the leaves ($Leaf(op, val)$) and the internal nodes ($Node(op, c_1, \ldots, c_n)$). In the following consider the sentence $a - 4 + c$.

1. Let's built an SDD with actions permitting to derive the syntax tree for expressions grammar in the form suitable for LR parsing.
   $E \rightarrow E_1 + T$, $E \rightarrow E_1 - T$, $E \rightarrow T$, $T \rightarrow (E)$, $T \rightarrow \textbf{id}$, $T \rightarrow \textbf{num}$
2. Let's repeat the exercise for an expression grammar parsable by LL parsers.
   $E \rightarrow TE'$, $E' \rightarrow +TE'_1$, $E' \rightarrow -TE'_1$, $E' \rightarrow \epsilon$, $T \rightarrow (E)$, $T \rightarrow \textbf{id}$, $T \rightarrow \textbf{num}$

Towards type checking

Let's now consider the case of a grammar for type definition:
$T \rightarrow BC$, $B \rightarrow \textbf{int}$, $B \rightarrow \textbf{float}$, $C \rightarrow [\textbf{num}]C$, $C \rightarrow \epsilon$
Define sematics rules to assign a type to an expression and try it on the sentence:

$$\textbf{int}[2][3]$$

ToC

Syntax Directed Translation

Syntax Directed Translation

A Syntax Directed Translation scheme permits to embed program fragments, called semantic actions, within production bodies. An SDT is a context-free grammar with program fragments embedded within production bodies.

- SDTs are an alternative approach to SDDs
- an STD is like an SDD except that the order of evaluation of the semantic rules is explicitly specified
- program fragments embedded within productions in curly braces are called semantic actions:

$$\mathit{rest} \rightarrow + \ \mathit{term}\ \{\text{print}('+')\}\ \mathit{rest}_1$$

Syntax Directed Translation

Construction

Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order.

However, SDT are typically implemented during parsing without the need to build a parse tree:

- introduce distinct marker nonterminals M_i in place of each embedded action;
- each marker has only one production $M_i \to \epsilon$.
- If the grammar with marker nonterminals can be parsed by a given method, then the SDT can be implemented during parsing

Syntax Directed Translation

STDs can be easily used to implement two important classes of SDDs:

- grammar LR-parsable and SDD S-attributed
- grammar LL-parsable and SDD L-attributed

In both cases the semantic rules of the SDD can be converted into an STD 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.

Postfix translation schemes

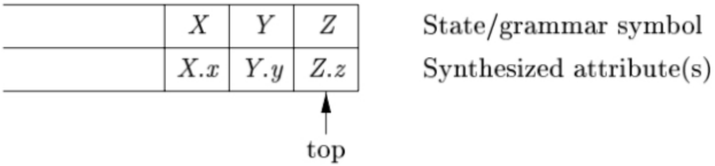
Simplest situation: bottom-up parsing with S-attributed SSD. In that case all the actions in the SDT are placed at the end of the production bodies. (Postfix SDT)

implementation

Postfix SDT are easy to implement with additional attributes for the stack cell. In particular it is useful to associate to each non-terminal on the stack the values assumed by the corresponding attributes.

Postfix translation schemes

For instance if you have a production like $A \rightarrow XYZ$ with a postfix SDT you will apply the actions in the SDT just before reducing XYZ to A.
Stack elements will be complex or include pointers to complex data structures.



SDT with actions inside productions

Consider the production $B \rightarrow X\{a\}Y$. When do we perform the action inside the production?

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

Imagine each SDT fragment as a distinct non-terminal M with the only production $M \rightarrow \epsilon$

Implementing SDT

Not all SDT can be implemented during parsing

General implementation rules

Any SDT can be implemented as follows:

- Ignore the actions and parse the input to produce a parse tree
- Examine each interior node, say a production $A \rightarrow \alpha$. 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
- Perform a preorder traversal of the tree, and as soon as a node labeled by actions is visited, perform that action

Implementing SDT

Not all SDT can be implemented during parsing

General implementation rules

Any SDT can be implemented as follows:

- Ignore the actions and parse the input to produce a parse tree
- Examine each interior node, say a production $A \to \alpha$. 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
- Perform a preorder traversal of the tree, and as soon as a node labeled by actions is visited, perform that action

SDT and Top-Down parsing

Note: Including semantic actions in grammars conceived for being parsable by top-down strategies can be complicated

Question: Would it be possible to define semantic actions and then transform the grammar?

Eliminating Left Recursion (simple case)

- In case included actions just need to be performed in the same order then it is enough to treat them as terminal symbols
  $E \to E + T\{print('+');\}$
  $E \to T$

When an SDT computes attributes we need to be more careful.

Eliminating Left Recursion (general case)

It is always possible to transform a recursive grammar with actions if it is S-attributed.
In particular given the grammar with actions:

$$A \to A_1 Y \quad \{A.a = g(A_1.a, Y.y)\}$$
$$A \to X \quad \{A.a = f(X.x)\}$$

Consider the parse tree fragment for a derivation:

$$\ldots A \ldots \xrightarrow{*} \ldots XYY \ldots$$

It is possible to rewrite it in an equivalent one according to the following schema:

$$A \to X \quad \{R.i = f(X.x)\} \quad R \quad \{A.a = R.s\}$$
$$R \to Y \quad \{R_1.i = g(R.i, Y.y)\} \quad R_1 \; \{R.s = R_1.s\}$$
$$R \to \epsilon \quad \{R.s = R.i\}$$

Eliminating Left Recursion (general case)

It is always possible to transform a recursive grammar with actions if it is S-attributed.
In particular given the grammar with actions:

$$\begin{array}{l} A \to A_1 Y \quad \{A.a = g(A_1.a, Y.y)\} \\ A \to X \quad \{A.a = f(X.x)\} \end{array}$$

Consider the parse tree fragment for a derivation:

$$\dots A \dots \overset{*}{\to} \dots XYY \dots$$

It is possible to rewrite it in an equivalent one according to the following schema:

$$\begin{array}{l} A \to X \quad \{R.i = f(X.x)\} \quad R \quad \{A.a = R.s\} \\ R \to Y \quad \{R_1.i = g(R.i, Y.y)\} \quad R_1 \ \{R.s = R_1.s\} \\ R \to \epsilon \quad \{R.s = R.i\} \end{array}$$

SDT for L-attributed definitions

Assuming a pre-order traversal of the parse tree we can transform a L-attributed SDD in a SDT as follows:

1. action computing inherited attributes must be computed before the occurrence of the non terminal. In case of more inherited attributes for the same non terminal order them as they are needed
2. actions for computing synthesized attributes go at the end of the production

Example

Consider the production:

$$S \rightarrow \textbf{while}\ (C)\ S_1$$

assuming the “traditional” semantics for this statement let’s generate the intermediate code assuming a three-address code where three control flow statements are generally used:

- `ifFalse` x `goto L`
- `ifTrue` x `goto L`
- `goto L`

Example

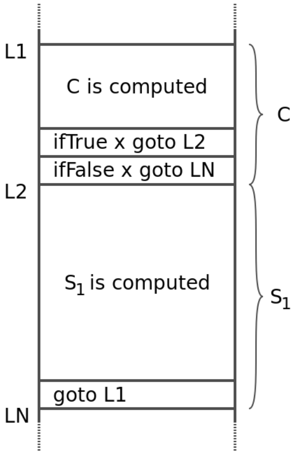
Consider the production:

$$S \rightarrow \textbf{while}\ (C)\ S_1$$

assuming the “traditional” semantics for this statement let’s generate the intermediate code assuming a three-address code where three control flow statements are generally used:

- `ifFalse` x `goto L`
- `ifTrue` x `goto L`
- `goto L`

Intermediate Code Structure



while statement - rationale

The following attributes can be used to derive the translation:

- *S.next*: labels the beginning of the code to be executed after *S* is finished
- *S.code*: sequence of intermediate code steps that implements the statement *S* and ends with *S.next*
- *C.true*: label for the code to be executed if *C* is evaluated to true
- *C.false*: label for the code to be executed if *C* is evaluated to false
- *C.code*: sequence of intermediate code steps that implements the condition *C* and jumps to *C.true* or to *C.false* depending on the evaluation

while statement - SDD and SDT

SDD

$S \rightarrow \textbf{while}\ (C)\ S_1$

$$
\begin{aligned}
&L1 = new();\\
&L2 = new();\\
&S_1.next = L1;\\
&C.false = S.next;\\
&C.true = L2\\
&S.code = \textbf{label} || L1 || C.code || \textbf{label} || L2 || S_1.code
\end{aligned}
$$

while statement - SDD and SDT

Note for the translation:

- L_1 and L_2 can be treated as synthesized attributes for dummy nonterminals and can be assigned to the first action in the production

SDT

$S \rightarrow$ **while** (	$\{L1 = new(); L2 = new(); C.false = S.next;$ $C.true = L2; \}$
$C)$	$\{S_1.next = L1; \}$
S_1	$\{S.code =$ **label** $\|\| L1 \|\| C.code \|\|$ **label** $\|\| L2 \|\| S_1.code\}$

while statement - SDD and SDT

Note for the translation:

- L_1 and L_2 can be treated as synthesized attributes for dummy nonterminals and can be assigned to the first action in the production

SDT

$$
\begin{array}{ll}
S \rightarrow \textbf{while} \ (& \{L1 = new();\ L2 = new();\ C.false = S.next; \\
 & C.true = L2;\} \\
C) & \{S_1.next = L1;\} \\
S_1 & \{S.code = \textbf{label} || L1 || C.code || \textbf{label} || L2 || S_1.code\}
\end{array}
$$

Implementing L-attributed SDD

Translation can be performed according to two different strategies:

- traversing a parse tree
- during parsing

Traversing a parse tree

- Build the parse tree and annotate; if the SDD is not circular there is at least an order of execution that works
- Build the parse tree, add actions, and execute the actions in preorder; e.g. L-attributed SDDs translated into SDTs

During parsing

- Use a recursive descent parser
- Generate code on the fly
- Implement an SDT in conjunction with an LL-parser
- Implement an SDT in conjunction with an LR-parser

Implementing L-attributed SDD

Translation can be performed according to two different strategies:

- traversing a parse tree
- during parsing

Traversing a parse tree

- Build the parse tree and annotate; if the SDD is not circular there is at least an order of execution that works
- Build the parse tree, add actions, and execute the actions in preorder; e.g. L-attributed SDDs translated into SDTs

During parsing

- Use a recursive descent parser
- Generate code on the fly
- Implement an SDT in conjunction with an LL-parser
- Implement an SDT in conjunction with an LR-parser

Implementing L-attributed SDD

Translation can be performed according to two different strategies:

- traversing a parse tree
- during parsing

Traversing a parse tree

- Build the parse tree and annotate; if the SDD is not circular there is at least an order of execution that works
- Build the parse tree, add actions, and execute the actions in preorder; e.g. L-attributed SDDs translated into SDTs

During parsing

- Use a recursive descent parser
- Generate code on the fly
- Implement an SDT in conjunction with an LL-parser
- Implement an SDT in conjunction with an LR-parser