4. Semantic Analysis I

Syntax Directed Definitions – Syntax Directed Translation Schemes

Andrea Polini

Formal Languages and Compilers
Master 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

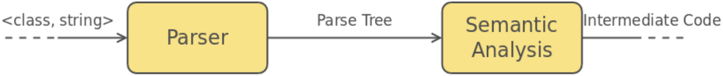
There are many additional checks that can be performed to increase correctness of code:

- 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:

$L \rightarrow E \quad E \rightarrow E + T \quad E \rightarrow T \quad T \rightarrow T * F \quad T \rightarrow F \quad F \rightarrow (E) \quad F \rightarrow id$

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 cycle 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:

$D \to TL; \quad T \to \textbf{int} | \textbf{float} \quad L \to L_1, \textbf{id} | \textbf{id}$

Let's add sematic rules to successively permit type checking

Exercise

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

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

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

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

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 S-attributed SDD to make the translation in decimal numbers

Construction of syntax trees using SDT

Syntax Tree

A syntax tree represents the hierarchical syntactic structure of the source program. Internal nodes are labeled with operator of the language while leaves are labeled with atomic element in the language. Syntax trees are useful for translation purpose making the phase much easier.

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.
   $E \rightarrow TE'$, $E' \rightarrow +TE'_1$, $E' \rightarrow -TE'_1$, $E' \rightarrow \epsilon$, $T \rightarrow (E)$, $T \rightarrow \textbf{id}$, $T \rightarrow \textbf{num}$

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.

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. SDT are typically implemented during parsing without the need to build a parse tree.

Particularly interseting are the cases:

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

Postfix translation schemes

Simplest situation: bottom-up parsing with S-attributed SSD. In that case all the actions in the SDT will follow 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 the non-terminal the values assumed by corresponding attributes.

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 appers 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)

SDT and Top-Down parsing

Note: Including semantic actions in grammars conceived for being parsable by top-down strategies is cumbersome

Question: Would it be possible to define sematic 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.g. $E \rightarrow E + T\{print('+');\} \quad E \rightarrow T$)

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:

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

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

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

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:

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

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

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

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`

Intermediate Code Structure

Example

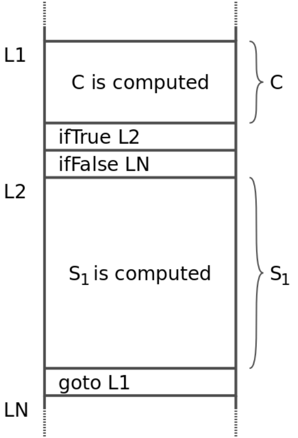
Consider the production:

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

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* of to *C.false* depending on the evaluation

while statement - SDD and SDT

SDD

$S \rightarrow$ **while** $(C)\ S_1$

$L1 = new();$
$L2 = new();$
$S_1.next = L1;$
$C.false = S.next;$
$C.true = L2$
$S.code = \textbf{label}||L1||C.code||\textbf{label}||L2||S_1.code||goto\ S1.next$

SDT

$S \rightarrow$ **while** ($\{L1 = new(); L2 = new(); C.false = S.next;$
$C.true = L2; \}$
$C)$ $\{S_1.next = L1; \}$
S_1 $\{S.code = \textbf{label}||L1||C.code||\textbf{label}||L2||S_1.code||goto\ S1.next\}$

while statement - SDD and SDT

SDD

$S \rightarrow \textbf{while}\ (C)\ S_1$	$L1 = new();$
	$L2 = new();$
	$S_1.next = L1;$
	$C.false = S.next;$
	$C.true = L2$
	$S.code = \textbf{label} \Vert L1 \Vert C.code \Vert \textbf{label} \Vert L2 \Vert S_1.code \Vert goto\ S1.next$

SDT

$S \rightarrow \textbf{while}\ ($	$\{L1 = new();\ L2 = new();\ C.false = S.next;$
	$C.true = L2;\}$
$C)$	$\{S_1.next = L1;\}$
S_1	$\{S.code = \textbf{label} \Vert L1 \Vert C.code \Vert \textbf{label} \Vert L2 \Vert S_1.code \Vert goto\ S1.next\}$