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:

$$rest \rightarrow + \; term \; \{\mathrm{print}('+')\} \; rest_1$$

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 $M_i \rightarrow \epsilon$.
- 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

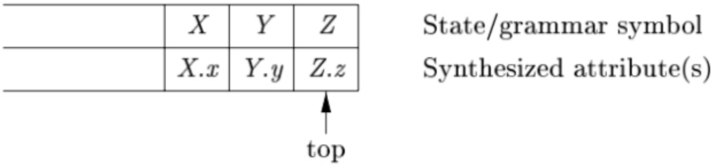
Simplest situation: bottom-up parsing with S-attributed SSD. In that case all the actions in the SDT are placed at the and of the production bodies. (Postfix SDT)

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 $B \to X\{a\}Y$. 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 $M \to \epsilon$

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 $A \rightarrow \alpha$. 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

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
  $E \rightarrow E + T\{print('+');\}$
  $E \rightarrow T$

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:

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

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:

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

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:

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

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:

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

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

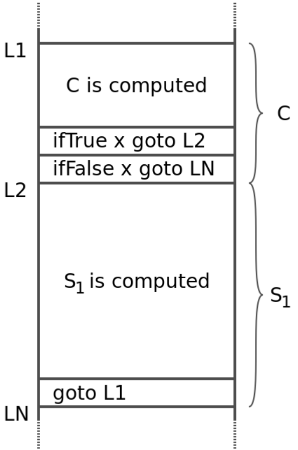
Consider the production:

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

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

while statement - SDD and SDT

SDD

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

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	$\{S.code = \textbf{label} \| L1 \| C.code \| \textbf{label} \| L2 \| S_1.code\}$

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