STYLES OF SYNTAX-DIRECTED TRANSLATIONS

SDT (Syntax-Directed Translation scheme)

Generates code while parsing, following rules associated with each production.

SDD (Syntax-Directed Definition)

Defines translation(s) associated with each node of the parse tree. Translation at root is the output.

Five Styles of Translation

1. Construct parse tree and perform a SDD.
2. Recursive procedures associated with nonterminals (like recursive-descent parser).
3. Recursive procedures that spit out code as a side-effect.
4. SDT on an LL grammar; top-down parse.
5. SDT on an LR grammar; bottom-up parse.

- (1) is completely general. (2) through (5) require an "L-attributed SDD."

Syntax-Directed Translations

The most common form of translator is an SDT, where the productions of a grammar have embedded within them certain *actions,* which cause side-effects.

Example:

The following grammar generates expressions over $+$ and $*$, while the SDT translates them into postfix.

$$\begin{aligned}
E &\rightarrow E + T \ \{\ \textbf{print}\ `+\text{'}\ \} \\
E &\rightarrow T \ \{\ \} \\
T &\rightarrow T * F \ \{\ \textbf{print}\ `*\text{'}\ \} \\
T &\rightarrow F \ \{\ \} \\
F &\rightarrow (E) \ \{\ \} \\
F &\rightarrow i \ \{\ \textbf{print}\ i.name\ \}
\end{aligned}$$

Here, $i.name$ is a translation of identifiers, found by taking the "lexical value" for i, which is a pointer to the appropriate symbol table entry, and obtaining from there the name of the identifier.

Order of Actions

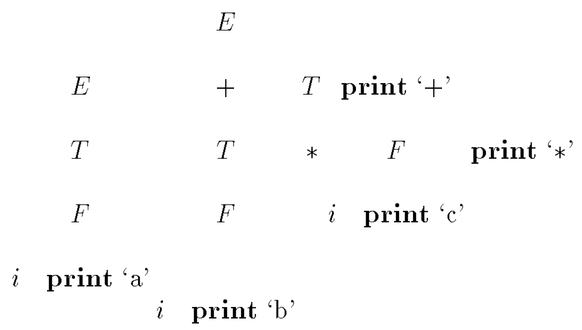
The interpretation of an SDT is that the action associated with a production is executed immediately after the terminals generated from any grammar symbols to its left are recognized.

Typical case: The parser is bottom-up, and all actions come at the end of the productions (called a *postfix* SDT). Then the actions take place when we reduce by their productions.

General rule: The actions are viewed as "dummy" terminals embedded in the parse tree. They are executed in the order of a preorder traversal of the tree. Thus, the result of the SDT is the same whether parsing is top-down, bottom-up, or something else.

Example:

Here is a parse tree from the previous grammar, with actions embedded as terminals. The input is $a+b*c$.



The output from a preorder traversal of this tree is $abc * +$.

- That is also the output obtained by executing the actions at the time of reduction during a bottom-up parse.
- The underlying grammar is not suitable for top-down parsing because of left-recursion.

Conversion to Top-Down Parsable SDT

The SDT notion is robust enough to allow conversion to a top-down parsable grammar, while we carry along the actions as though they were terminals. That is, the SDT performed is not changed by such a transformation, just as the input language accepted is unchanged.

Recall that we can eliminate left-recursion from the productions

$$A \rightarrow A\alpha \mid \beta$$

by replacing them with

$$\begin{aligned} A &\rightarrow \beta B \\ B &\rightarrow \alpha B \mid \epsilon \end{aligned}$$

Example:

Consider

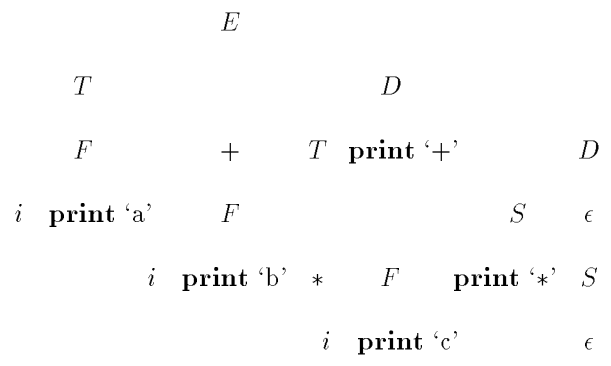
$$\begin{aligned} E &\rightarrow E + T \ \{ \ \textbf{print} \text{ '+' } \} \\ E &\rightarrow T \end{aligned}$$

Here, $\alpha = + \, T \ \{ \ \textbf{print} \text{ '+' } \}$ and $\beta = T$. The new rules for the SDT are thus:

$$\begin{aligned} E &\rightarrow T \ D \\ D &\rightarrow + T \ \{ \ \textbf{print} \text{ '+' } \} \ D \\ D &\rightarrow \epsilon \end{aligned}$$

The fact that the action is embedded within the second production is not, in principle, a problem. It says that immediately after seeing on the input a string consisting of a +-sign and a string that parses to a term (T), we emit the symbol +, before looking for another expression tail (D).

If we make a similar transformation for nonterminal T, we obtain an LL grammar for expressions, whose parse tree, with embedded actions, for the input $a + b * c$ is:



Notice that a preorder traversal of this tree also yields output $abc*+$.

Syntax-Directed Definitions

When translating according to an SDT, it is normal not to construct the parse tree. Rather we perform actions as we parse; that is, we "go through the motions" of building the parse tree.

To perform a translation that cannot be expressed by an SDT directly, it helps to build the parse tree first. For example, while we can translate infix to postfix by an SDT, we cannot translate to prefix.

Define one or more *attributes* of each terminal and nonterminal, and give rules relating the values of these translations at the various nodes of the parse tree.

- Rules can be associated with productions, because they relate translations at a node to translations at its parent and/or children.

The rules for computing these translations together form a *syntax-directed definition* (SDD).

- Rules with no side effects (such as printing output) are called an *attribute grammar*.

Example:

We can translate expressions to prefix form if we are allowed to associate a string (the prefix expression) with every node labeled E, T, or F.

The string for a node corresponding to the left side of a production is formed from the strings corresponding to the nonterminals on the right by concatenation, denoted $\big|$. Put another way, strings are computed bottom-up, from children to parents.

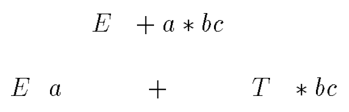
Thus, an SDD for infix-to-prefix translation is

$$\begin{array}{lll} E & \rightarrow & E_1 + T \; \{ \; E.pre := \\ & & \qquad \text{'+'} \big| E_1.pre \big| T.pre \; \} \\ E & \rightarrow & T \; \{ \; E.pre := T.pre \; \} \\ T & \rightarrow & T_1 * F \; \{ \; T.pre := \\ & & \qquad \text{'*'} \big| T_1.pre \big| F.pre \; \} \\ T & \rightarrow & F \; \{ \; T.pre := F.pre \; \} \\ F & \rightarrow & (E) \; \{ \; F.pre := E.pre \; \} \\ F & \rightarrow & i \; \{ \; F.pre := i.name \; \} \end{array}$$

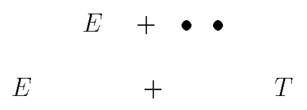
- Note that E_1 and T_1 are instances of nonterminals E and T, respectively; the "sub 1" is intended to remind the listener that the occurrence on the right is referred to.

Replacing Concatenation by Indirection

The above definition implies that as we go up the tree, a string is computed by copying at each node, so the root of the tree for $a + b * c$ would look like:



In practice, one would use pointers to structures representing the strings at the children, and each node would have a record with literals and pointers, as:



Synthesized Attributes

A SDD defines zero or more *attributes* for each nonterminal and terminal. A *synthesized attribute* has its value defined in terms of attributes at its children.

Example:

A production $A \rightarrow BC$ and a rule like

$$A.att := f(B.att1,\ B.att2,\ C.att3)$$

makes $A.att$ a synthesized attribute.

- It is conventional to call the attributes of terminals, which are generally lexical values returned by the lexical analyzer, "synthesized."
- A SDD with only synthesized attributes is called an *S-attributed definition*. All the examples seen so far are S-attributed.

Implementing S-attributed Definitions

It is easy to implement an S-attributed definition on an LR grammar by a postfix SDT.

- Values of attributes for symbol X are stored along with any occurrence of X on the parsing stack.
- When a reduction occurs, the values of attributes for the nonterminal on the left are computed from the attributes for the symbols on the right (which are all at the top of the stack), before the stack is popped and the left side pushed onto the stack, along with its attributes.

"Main Attributes"

It is common for there to be one "main" attribute, which is a long string, e.g., intermediate code, while other attributes are short "auxiliaries," e.g., types of expressions or labels of statements used for control flow in the intermediate code.

If the rule for the main attribute concatenates the main attributes of the nonterminals on the right side of the production, *in the same order*, perhaps with additional literals among them, then we can implement the SDD by an SDT whose actions are to emit the literals in their proper place (as well as computing all auxiliaries on the parsing stack).

Example:

The infix-to-postfix SDD involves one attribute for each nonterminal, which we may call *post*. The SDD has rules like:

$E.post := E_1.post \,\Big|\, T.post \,\Big|\, \text{`+'}$

which can be turned into SDT

$E \rightarrow E + T \; \{ \textbf{ print } \text{`+'} \}$

In principle, if we wanted to translate to prefix, we could blithely write the SDD with rules like:

$E.pre = \text{`+'} \,\Big|\, E_1.pre \,\Big|\, T.pre$

The corresponding SDT, with rules like

$E \rightarrow \{ \textbf{ print } \text{`+'} \} \; E + T$

is legal, but *cannot be executed as written*, because the grammar will not let the parser know when to execute the print actions. Rather, it must be implemented as discussed previously: build the parse tree and then traverse it in preorder to execute the actions.

Important Points:

- Notice that an SDT is an implementation of an SDD.
- The proper SDT to use for an SDD may depend on the type of parser used.
- If an SDD can be converted to a postfix SDT, where all actions take place at the ends of productions, and the underlying grammar is bottom-up parsable, then the SDD can be implemented without explicitly building the parse tree.
- If we must build the parse tree, we can do so with an SDT and either a top-down or a bottom-up parser (create nodes when reducing or expanding).

Example:

(This works bottom-up; see Dragon book for top-down construction.)

$E \rightarrow E_1 + T \; \{ E.node :=$
$\quad \text{getnode}(\text{`+'}, E_1.node, T.node) \}$

Note the action describes a manipulation of the parsing stack, where pointers to nodes of the parse tree are kept.

Inherited Attributes

Any attribute that is not synthesized is called *inherited.*

- The typical inherited attribute is computed at a child node as a function of attributes of its parent.
- It is also possible that attributes at sibling nodes (including the node itself) will be used.

Example:

Consider the grammar with nonterminals

1. D = type definition.
2. T = type (integer or real).
3. L = list of identifiers.

and SDD

$D \rightarrow T\ L$

$\qquad L.type := T.type$

$T \rightarrow int$

$\qquad T.type :=$ INT

$T \rightarrow real$

$\qquad T.type :=$ REAL

$L \rightarrow L_1\ ,\ id$

$\qquad L_1.type := L.type;$

$\qquad$ addtype($id.entry,\ L.type$)

$L \rightarrow id$

$\qquad$ addtype($id.entry,\ L.type$)

Notes

- The call to *addtype* is a side-effect of the SDD. The timing of the call is not clear, but it must take place at least once for every occurrence of the last two productions in the parse tree.
- We shall discuss "L-attributed definitions," where there is a natural order of evaluation, making ambiguities like this one disappear.
- We assume that terminal *id* has an attribute *entry*, which is a pointer to the symbol table entry for that identifier. The effect of *addtype* is to enter the declared type for that identifier.

Circularity and Order of Evaluation

When we have inherited attributes, the order of events becomes unclear, and we frequently must create the parse tree, then follow the rules for computing the attributes at the various nodes.

When so doing, we may have to visit the same node many times, and the process may not even converge.

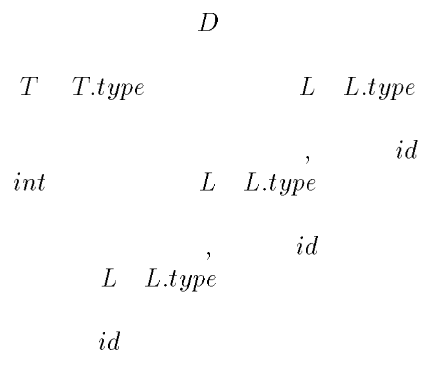
- The Dragon book discusses *dependency graphs*. These are defined for each parse tree, and show how attributes at various nodes depend on one another.
- Cycles in the dependency graph for *any* parse tree indicates the whole SDD is defective.

Example:

For the input

```
int a, b, c
```

The parse tree and dependencies, according to the SDD for types given above, are



- Note there are no cycles in *this* parse tree. To show the SDD is noncircular, we need to show that about *every* parse tree.

Example: Simplifying Code Generation for Control Flow by Using Inherited Attributes

We shall now work one example in five different translation styles. The example concerns the translation of while-statements.

While we can translate common control constructs (while, if, etc.) with synthesized attributes only, by the technique known as "backpatching," it is handy to use inherited attributes to pass down the parse tree certain labels representing targets of jumps.

Style I: General SDD

Our SDD rule for while-statements is

```
$S \rightarrow$ while $C$ do $S_1$
    $L_1$ := new();
    $L_2$ := new();
    $S_1.next$ := $L_1$;
    $C.false$ := $S.next$;
    $C.true$ := $L_2$;
    $S.code$ := 'label' $\mid L_1 \mid C.code \mid$
        'label' $\mid L_2 \mid S_1.code$
```

- S (statement) has synthesized attribute *code* (code to execute the statement) and inherited attribute *next* (label to go to after executing code).
- C (condition) has synthesized attribute *code* (to test the condition) and inherited attributes *true* and *false* (the labels to go to if the condition is found true or false, respectively).
- *new*() is a procedure that produces a new label each time called.

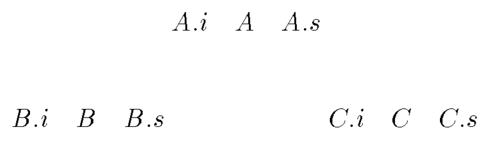
- Other types of statements must have rules that follow the same discipline, producing appropriate values for these attributes.
- This SDD can be executed directly if we build the parse tree, then execute the rule (and similar rules for other language parts) at each node in the parse tree in a greedy order; that is, statements are executed at a node as soon as the values on which they depend have been computed at themselves or other nodes.

L-Attributed Definitions

A SDD is *L-attributed* if all attributes are either

1. Synthesized,
2. *Extended* synthesized attributes, which can depend not only on attributes at the children, but on inherited attributes at the node itself, or
3. Inherited, but depending only on inherited attributes at the parent and any attributes at siblings to the left.

For L-attributed SDD's, there is a natural order in which to evaluate the attributes attached to a parse tree. traverse the tree in preorder, computing inherited attributes the first time a node is reached and synthesized attributes the last time it is reached.



Style II: Recursive Descent

If the underlying grammar is LL, we can design a "recursive descent" parser that consists of a collection of recursive procedures, one for each nonterminal.

The procedure A for nonterminal A takes as arguments all the inherited attributes of A, and it returns all the synthesized attributes of A, which we assume for convenience are bundled into one synthesized attribute.

Example (While-Statements):

```
procedure $S(next)$;
string $Ccode,\ Scode$; /* store returned values
    $C.code$ and $S_1.code$ locally */
label $L_1,\ L_2$;
begin
    if (current input = 'while') then begin
        advance input;
        $L_1 :=$ new();
        $L_2 :=$ new();
        $Ccode := C(next,\ L_2)$;
        check 'do' on input and advance;
        $Scode := S(L_1)$;
        return('label' $\|$ $L_1$ $\|$ $Ccode$ $\|$
            'label' $\|$ $L_2$ $\|$ $Scode$)
    end
    else /* other statement types */
```

- We do not actually return a concatenation of strings, but rather a structure with pointers to represent the desired result.

Style III: On-The-Fly Code Generation

If each nonterminal, like S and C, has a "main" attribute *code*, we can generate code on the fly, rather than as a return value.

- In each production the *code* for the left side must consist of literals and the *code* for the nonterminals on the right side, in order.

Thus, we may write the procedure S as

```
procedure $S(next)$;
begin
    if (current input = ‘while’) then begin
        advance input;
        $L_1$ := new();
        $L_2$ := new();
        print ‘label’; print $L_1$;
        $C(next,\ L_2)$;
        print ‘label’; print $L_2$;
        check input = ‘do’ and advance;
        $S(L_1)$
    end
    else /* etc. */
```

Style IV: SDT on an LL Grammar

We can convert an L-attributed SDD into an SDT if we

1. Place the computation of inherited attributes for a nonterminal immediately before that nonterminal in right sides of productions.
2. Place the computation of synthesized attributes at the ends of productions.

- If the underlying grammar is LL, then we can parse top-down, and the SDT will “work.”
- If we execute actions during an LL parse, $A.i$ will be available just before we expand A at the top of the stack, and $A.s$ will be produced at the point where A was on the stack.
- Store $A.i$ with A on the stack, and put $A.s$ with the symbol below A on the stack.

Example:

The previous SDD could be converted into an SDT with rules like:

```
$S$ → while
    {   $L_1$ := new();
        $L_2$ := new();
        print ‘label’; print $L_1$;
        $C.false$ := $S.next$;
        $C.true$ := $L_2$;
    }
    $C$
    do
    {   print ‘label’; print $L_2$;
        $S_1.next$ := $L_1$;
    }
    $S_1$
```

Locating Attributes on the Parser Stack

If the underlying grammar is LL, the above SDT makes sense, but we have to be careful where we store values associated with the S on the left, as the parser stack is manipulated.

For example, suppose $S.next$ is available along with S on the stack. When that S is expanded, any of its actions could, in principle, need the value $S.next$.

- If we are clever, we can observe that the only thing done with $S.next$ is that it becomes the value of $C.false$, and immediately place that value in the stack cell for C as we expand S.
- More generally, we would store $S.next$, like all translations of the expanded nonterminal, in the stack cells representing each of the actions in the rule.
- Then, the first action would copy $S.next$ into the cell below it, as the value of $C.false$. The second action would not use its copy of $S.next$.

Style V: SDT on an LR Grammar

- It has long been supposed that LL parsing made up for the fact that the class of suitable grammars was smaller, by being more versatile for translation than LR parsing.
- In fact that is not the case.
- Whenever we have an LL grammar and an L-attributed translation scheme, we can insert *marker* nonterminals into the right sides of productions so that all inherited and (if needed) synthesized attributes can be found on the bottom-up parsing stack a fixed distance away from the nonterminal owning those attributes.
- The general rules are found in Section 5.6 of the Dragon Book. We shall only give the intuitive idea.

Moving Actions to the End of Productions

One essential for bottom-up parsing is that actions take place only at reduction time, so all actions must be at the right end of productions. Thus, if we have a production like:

$$A \rightarrow B \; \{ \text{ action } \} \; C$$

we introduce a marker M, rewriting the SDT as

$$\begin{aligned} A &\rightarrow BMC \\ M &\rightarrow \epsilon \; \{ \text{ action } \} \end{aligned}$$

- If the original grammar is LL, the new one will be LR, but if the old were only LR, the new might not be LR.

Keeping Inherited Attributes Immediately Below Their Nonterminal

The second necessary trick is to keep each inherited attribute of some nonterminal, say A, immediately below A on the stack; that is, it is associated with the grammar symbol immediately to the left of A in a sentential form.

- We must have these attributes available *before* we reduce to A, in fact, immediately before we start to reduce an input substring to A.
- It is possible to keep the needed attributes more than one position below that of A on the stack, but the position must not depend on what is below A on the stack.

Example:

Suppose we have production $A \rightarrow BC$. Inherited attributes for B can only depend on inherited attributes of A. If we have a rule $B.i := f(A.i)$, we can introduce a marker M with rules:

$$\begin{aligned} A &\rightarrow MBC \\ M &\rightarrow \epsilon \\ &\quad \{ \; M.i := A.i; \; M.s := f(M.i) \; \} \end{aligned}$$

Now $B.i$ is available on the stack (as $M.s$) immediately below where B will eventually appear, even though we have just now begun to recognize B.

Similarly, an inherited attribute of C can depend on inherited attributes of A and all attributes of B.

- Inherited attributes of A can be copied to a new marker N, preceding C.
- Attributes of B will appear with B on the stack before we begin the recognition of C. They also can be copied to N when we reduce ϵ to N.

Example:

The while-statement can be handled bottom-up by the following SDT. Note that L_1 and L_2 are regarded here as synthesized attributes of the marker M.

```
$S \rightarrow$ while $M$ $C$ do $N$ $S_1$

$M \rightarrow \epsilon$
    {   $L_1$ := new();
        $L_2$ := new();
        $C.true$ := $L_2$;
        $C.false$ := $S.next$;
        print 'label'; print $L_1$;
    }

$N \rightarrow \epsilon$
    {   $S_1.next$ := $L_1$;
        print 'label'; print $L_2$;
    }
```

Before we find **while** on the input, the top of the parsing stack has, by the inductive hypothesis, the inherited attribute $S.next$ for the statement we are about to recognize. We shift the **while**, leaving

$\cdots [S.next]$ **while**

Then, we reduce ϵ to M, executing the action associated with that reduction. It involves the computation of L_1, L_2, $C.true$, and $C.false$, all of which are associated with that M on the stack.

$\cdots [S.next]$ **while** $M[L_1, L_2, C.true, C.false]$

To compute $C.false$, we had only to reach into the stack, find $S.next$, and copy it.

We now have the inherited attributes of C sitting on top of the stack and are ready to recognize C. After we do so, we push C onto the stack, the keyword **do** onto the stack, and reduce ϵ to N.

The action for N requires us to find L_1, which is at the level of M, the third below the top. That value becomes $S_1.next$, which we place on top of the stack and we are ready to recognize S_1.

$\cdots [S.next]$ **while** $M[L_1, L_2, C.true, C.false]$
C **do** $N[S_1.next]$