Syntax-Directed Translation

Read Chapter 5 of the Dragon Book (you can skip sections 5.5, 5.6, 5.7, 5.8, 5.9, and 5.10).

Introduction

After syntax analysis, the next phase in a compiler is semantic analysis. Semantic analysis, in turn, is followed by intermediate code generation. In these lecture notes, we are going to do both semantic analysis and intermediate code generation by computing various kinds of values in parse trees using what is called syntax-directed translation. The goal of these lecture notes is therefore to present the basics of syntax-directed translation so that we can later use this knowledge when explaining semantic analysis and intermediate code generation.

In syntax-directed translation, each grammar symbol (token or nonterminal) in a context-free grammar can have multiple *attributes*. An attribute is like a normal C or Java program variable (it can be read or written) except that it is attached to a specific grammar symbol.

To compute attributes, we use *attribute computation functions*, also called *semantic rules*. Each production in a context-free grammar has zero or more semantic rules that specify how to compute the attributes for the corresponding grammar symbols.

There are two different ways to write semantic rules in syntax-directed translation: *syntax-directed definitions* and *translation schemes*. The difference between the two is that translation schemes specify precisely the order in which attributes must be computed, while syntax-directed definitions do not. In these lecture notes we will only look at syntax-directed definitions, since, with the exception of one semantic rule in the intermediate code generation, we will not need the extra specification power of translation schemes.

There are also two different kinds of attributes: *synthesized* attributes are computed bottom-up in a parse tree, while *inherited* attributes can be computed top-down and left-right and right-left. For a grammar production of the form $A \to X_1 \dots X_n$, a semantic rule to compute a synthesize attribute has the following form:

$$A.a := f(X_1.x_1, \dots, X_n.x_n)$$

so the attribute a of the grammar symbol A depends only on the attributes $x_1, \dots, x_n$ of the grammar symbols $X_1, \dots, X_n$ of the RHS of the grammar production. This means that, in a parse tree, the attribute a of A only depends on the attributes of the children of A in the tree, so the computation can indeed be done bottom-up. For the same grammar production, a semantic rule to compute an inherited attribute has the following form:

$$X_i.x_i := f(A.a, X_1.x_1, \ldots, X_{i-1}.x_{i-1}, X_{i+1}.x_{i+1}, \ldots, X_n.x_n)$$

so the attribute x_i of the grammar symbol X_i depends not only on all the attributes of the other grammar symbols in the RHS of the grammar production but also on the attribute a of the LHS A. This means that, in a parse tree, the attribute x_i of X_i depends both on the attributes of the siblings $X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n$ of X_i as well as on the attribute a of the parent A of X_i. In that case it is less clear in which order the attributes in the parse tree should be computed (more on this later).

When an attribute is synthesized, the corresponding bottom-up computation has to start at the leaves of the parse tree, where tokens are. Since such a bottom-up computation needs initial values to start, the values of the synthesized attributes for the tokens at the leaves of a parse tree need to come from somewhere else, they cannot be the result of the bottom-up computation itself, so they cannot be the result of a semantic rule (which should be obvious, since, based on the shape of the semantic rules for synthesized attributes that we described just above, such rule can only have an attribute for a nonterminal as a result). Synthesized attributes for tokens are then called *intrinsic*, meaning that their values are not computed by the semantic rules for the attributes.

In both the synthesized and inherited cases, the attribute computation function f can be in fact either a function or a procedure. Remember (or learn) that a function computes a value and does not have side effects (i.e. the state of the computer is not modified by the computation) while a procedure does not compute a value and has only side effects (i.e. a procedure only changes the state of the computer). If all the semantic rules in syntax-directed definitions are functions then the syntax-directed definitions and the corresponding grammar productions together form an *attribute grammar*. So an attribute grammar is just a special case of syntax-directed definitions for which the semantic rules are restricted to using functions only, not procedures.

If all the attributes used in semantic rules are synthesized, then the syntax-directed definitions are said to be *S-attributed* definitions. In other word, S-attributed definitions are syntax-directed definitions for which all the attributes are synthesized. Such S-attributed definitions are of interest because all the attributes in such definitions can then be computed bottom-up in a parse tree, which in turn means that it will be easy to compute the attributes using an LR (bottom-up) parser.

Examples

We now look at three examples that illustrate the use of both synthesized and inherited attributes in semantic rules, as well as the use of procedures.

Expression Values

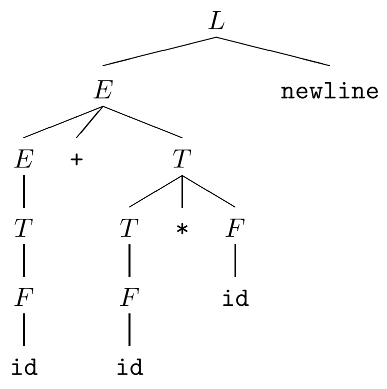
Here is an example of S-attributed definitions that compute and print the value of an arithmetic expression:

$L \to E$ `newline`	print(E.val)
$E_1 \to E_2$ `+` T	E_1.val := E_2.val + T.val
$E \to T$	E.val := T.val
$T_1 \to T_2$ `*` F	T_1.val := T_2.val $*$ F.val
$T \to F$	T.val := F.val
$F \to$ `(` E `)`	F.val := E.val
$F \to$ `id`	F.val := `id`.val

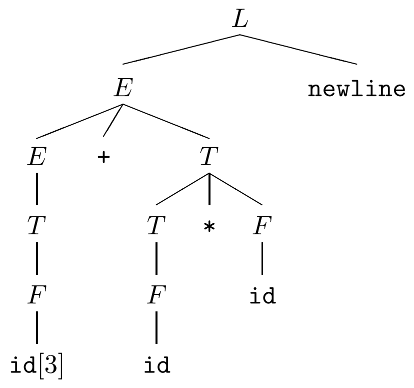
Here the "val" attribute is used to store the value associated with the corresponding grammar symbol. The semantic rule associated with each grammar production simply implement the corresponding arithmetic computation. There are a few things to note here.

- The nonterminal E appears twice in the second grammar production, and the nonterminal T appears twice in the fourth grammar production. We therefore have to give extra indices to these nonterminals so we are able to differentiate them when writing the corresponding semantic rules.
- The `+` and `*` grammar symbols are tokens that come from the compiler's lexical analyzer. The $+$ and $*$ in the the semantic rules are the usual mathematical addition and multiplication operators. The former are purely syntax, while the latter are purely semantics: the mathematical operator $+$ gives the meaning of the syntax `+` and the mathematical operator $*$ gives the meaning of the syntax `*`. Do not confuse syntax and semantics when writing syntax-directed definitions: the grammar production on the left are syntax, the semantic rules on the right are meaning.
- If you look at the direction in which information flows in each semantic rule, you will see that it always flows from the attributes of grammar symbols in the RHS of the corresponding grammar production to the attribute of the grammar symbol in the LHS of the corresponding production. In other words, information always flows bottom-up in the corresponding piece of parse tree. In turn this means that the val attribute is synthesized, and the semantic rules above are indeed S-attributed definitions.
- The semantic rules above use `id`.val but never define it. This is because `id` is a token and the synthesized attributes of tokens are always intrinsic: by definition the values of such attributes do not come from the semantic rules themselves but from somewhere else. In the rules above, the value of `id`.val will be computed by somehow looking at the value stored in the variable represented by the token `id`.

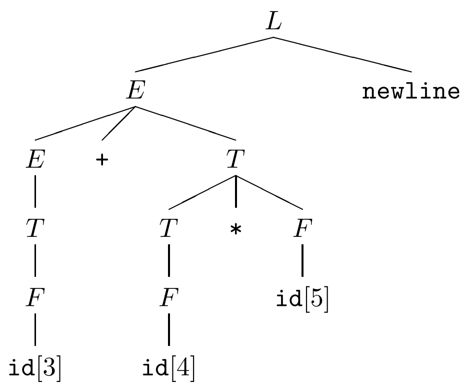
Here is an example of how to use these semantic rules. Assume that the source code given to the compiler is: `x + y * z \n`, where `\n` represents a newline character (the character that you type in the source code when you press the Enter or Return key on your keyboard). The lexical analyzer transforms the source code into the following sequence of tokens: `id + id * id newline`. The syntax analysis then creates the following parse tree:



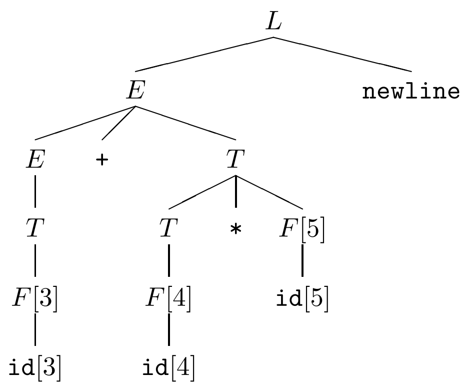
We can then use the semantic rules above to compute the val attributes of the different grammar symbols in that parse tree. Since the val attribute is synthesized, we know that doing the computation in any bottom-up order will give us the correct result. To start the bottom-up computation though, we first have to compute the val attributes for the `id` leaves of the parse tree, for which the val attributes are intrinsic, as we indicated above. To compute the val attribute of the leftmost token `id` in the parse tree, for example, we have to know the value which is stored in the corresponding identifier. Since the leftmost `id` token corresponds to the lexeme `x`, the val attribute for that token is the value stored in variable `x`. Let's suppose that `x` is in fact a constant in the source program and that it's value is defined once for all to be 3. Then the compiler can determine that `id`.val is 3:



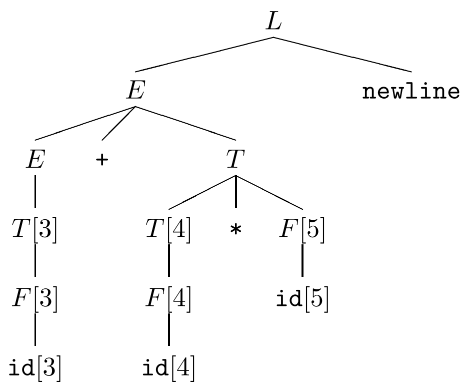
Here we represent the value of the val attribute by putting it between brackets []. Similarly, if `y` is a constant with known value 4 and `z` is a constant with known value 5, then we get the following tree:



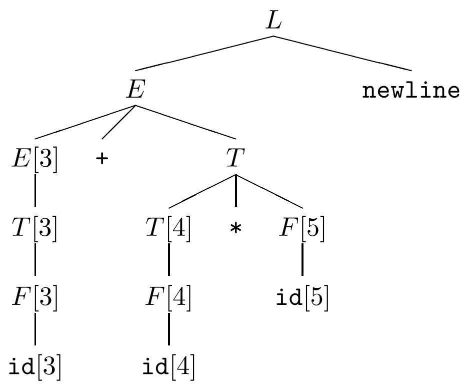
Note that so far we have not used any of the semantic rules above, since we have only been computing intrinsic attributes. Note also that the order in which these intrinsic attributes are computed does not matter. Once these intrinsic attributes have been computed, then we can start using our semantic rules above to compute the val attributes of the nonterminals in the parse tree. For example, we can use the semantic rule F.val := `id`.val three times to compute the val attributes for the three F nonterminals in the parse tree:



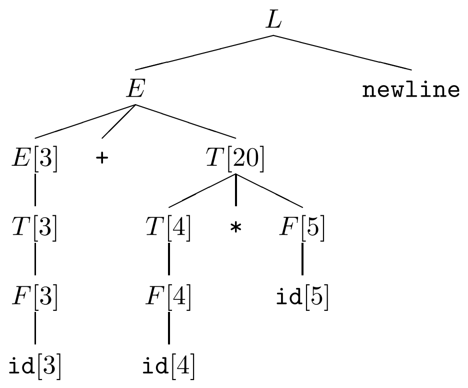
Again this can be done in any order, as long as it is done bottom-up. Next we can use, for example, the rule T.val := F.val twice to compute the val attributes for the two bottom-most T nonterminals in the parse tree:



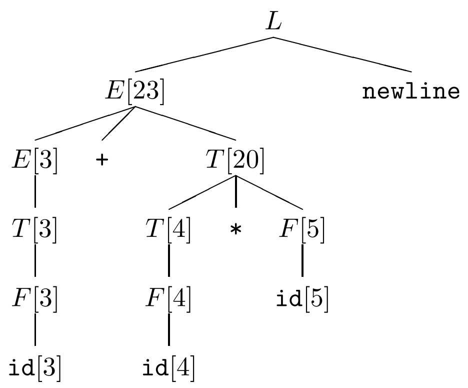
Next we can for example use the semantic rule $E.\text{val} := T.\text{val}$ to compute the val attribute of the bottom-most E in the parse tree:



After that there is no choice: we have to use the rule $T_1.\text{val} := T_2.\text{val} * F.\text{val}$ to compute the val attribute of the top-most T nonterminal. Here we already know that $T_2.\text{val}$ is 4 and that $F.\text{val}$ is 5, therefore $T_1.\text{val}$ is 20:



Similarly we use the semantic rule $E_1.\text{val} := E_2.\text{val} + T.\text{val}$ and the facts that $E_2.\text{val}$ is 3 and $T.\text{val}$ is 20 to compute that $E_1.\text{val}$ is 23:



Finally our bottom-up computation reaches the top of the parse tree. Here we use the semantic rule print(E.val). This rule does not compute a val attribute but instead just uses the print function to do a side-effect, which is to print the value of E.val (i.e. the number 23) to the output so that the user of the compiler can see that the value of the expression `x + y * z` is 23.

Variable Definitions

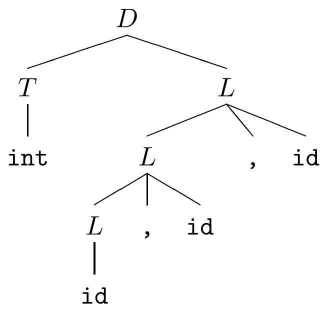
Here is another example of syntax-direction definitions, but this time using a combination of synthesized and inherited attributes. Therefore the following syntax-directed definitions are not S-attributes definitions (we will see later that they are what is called L-attributed definitions). These definitions are used to compute a type and propagate that type to the corresponding variables in a sequence of variable definitions like: `int x, y, z`. These syntax-directed definitions use two attributes: one attribute "syntype" which is a synthesized attribute to compute the type used in the variables definition, and one attribute "intype" which is an inherited attribute to propagate the type of the definition to the various variables.

$D \rightarrow T\ L$	L.intype := T.syntype
$T \rightarrow$ `int`	T.syntype := `int`.syntype
$T \rightarrow$ `float`	T.syntype := `float`.syntype
$L_1 \rightarrow L_2$ `, id`	$\begin{cases} L_2.\text{intype} := L_1.\text{intype} \\ \text{addtype}(\texttt{id}.\text{entry}, L_1.\text{intype}) \end{cases}$
$L \rightarrow$ `id`	addtype(`id`.entry, L.intype)

The nonterminal D here represents a variables definition: a type followed by a list of identifiers. As you can see from the first semantic rule, a definition, by itself, does not have a type so D does not have any attribute. Rather, the type coming from T is transferred to the list of variables L. Note how, in this first semantic rule, information does not flow from the RHS of the grammar production to its LHS, but instead information flows between symbols of the RHS. This proves that some of the attributes (intype, in the present case) are inherited.

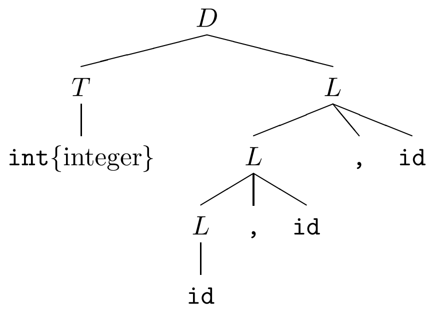
The fourth semantic rule above has to do two things: propagate the inherited attribute intype further down the parse tree (since information flows from L_1 in the LHS to L_2 in the RHS) and call the addtype procedure. This procedure takes as argument the entry attribute of the identifier token `id`, which is a pointer to the entry in the symbol table for that identifier, and the type which is to be assigned to the variable. Given the type and a pointer to an entry in the symbol table, the addtype procedure will simply store that type in that entry, therefore assigning that type to the corresponding variable. This also has to be done in the fifth semantic rule, the only difference being that there is no need to propagate the type further down the tree since the end of the list of variables (i.e. bottom of the parse tree) has been reached.

For example, consider the following variable definitions: `int x, y, z`. From this piece of source program, the lexer and parser will create the following parse tree:

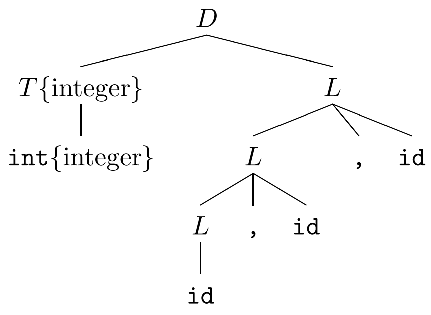


The first attribute we have to compute is the attribute syntype of `int`. Since `int` is a token and the attribute syntype is synthesized, we therefore know that syntype is in fact in this case an intrinsic attribute, so its value cannot be computed using the semantic rules above and the compiler has to find its value from somewhere else. In the present case, the value of the syntype attribute for an `int` token is simply defined once for all as a constant inside the code of the compiler, to be the type integer (similarly the syntype attribute of a `float` token is defined once for all to be floatingpoint). In other words, the meaning of the token `int` is the type integer.

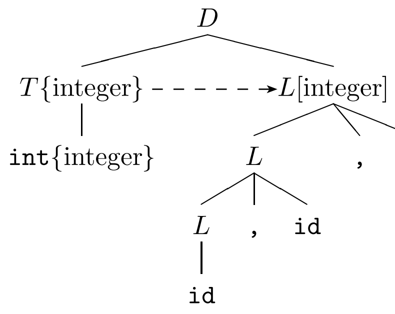
Here we represent the value of the syntype attribute by putting it between braces next to the grammar symbol:



Using the second semantic rule above, we then compute that the syntype attribute of the T is integer as well:

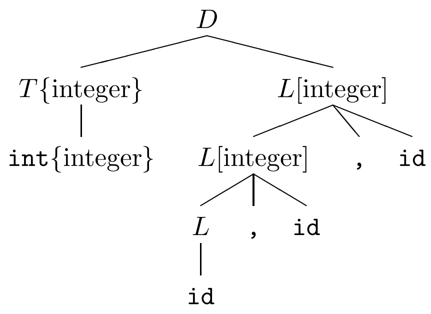


At the next step we use the first semantic rule to transfer the type information from the synthesize syntype attribute to the inherited intype attribute (the value of which we put between brackets in the parse tree):

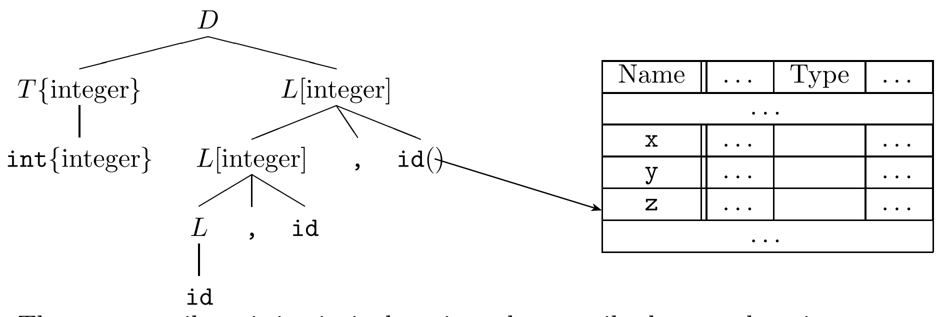


Note how, during this step, information does not flow bottom-up anymore, but flows sideways (see the dashed arrow), which proves that the intype attribute is not synthesized (i.e. it is inherited).

We then use the fourth semantic rule. In the first part of that rule, we propagate the type information down the right side of the parse tree:



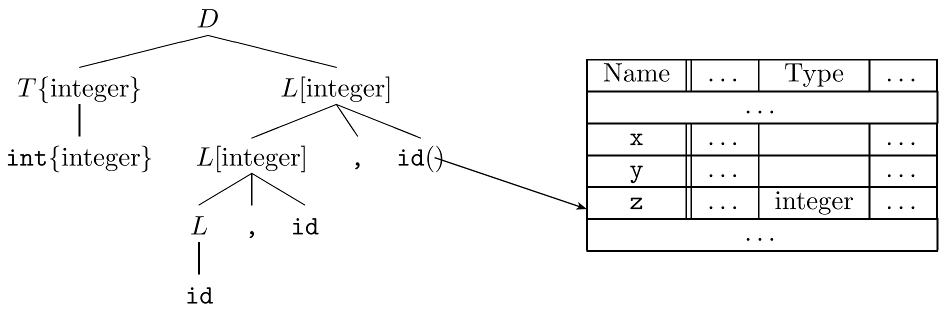
In the second part of the same rule, we compute the entry attribute of the identifier. The entry attribute (represented below between parentheses) is a pointer to the entry in the symbol table for the corresponding variable, which is the variable `z` (since the `id` token we are currently looking at in the parse tree is the leftmost one):



Name	...	Type	...
...			
x	...		...
y	...		...
z	...		...
...			

The entry attribute is intrinsic, but since the compiler knows where in memory its symbol table is, it then only has to search in the table for the entry for `z` and use the result as the value of the entry attribute.

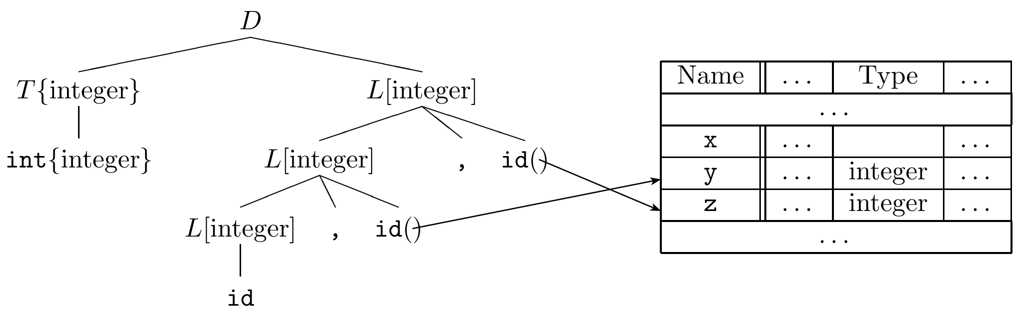
We then give this entry attribute of the identifier and the intype attribute for the upper L in the parse tree as arguments to the addtype procedure. Given this information, the addtype procedure changes the symbol table entry for `z` to store in it the type of that variable:



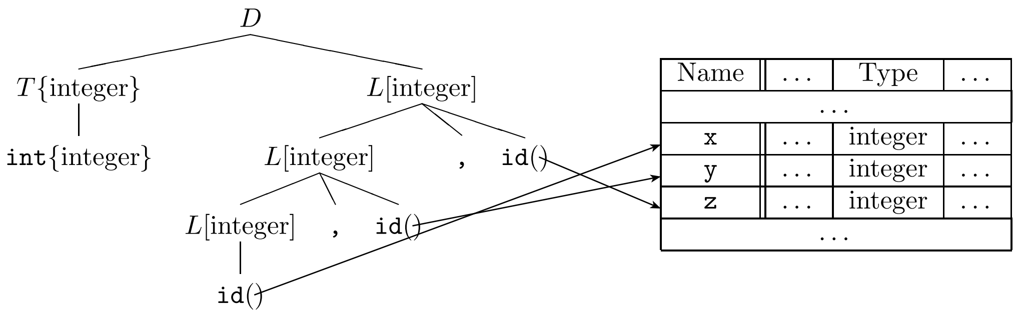
Name	...	Type	...
...			
x	...		...
y	...		...
z	...	integer	...
...			

This is how the compiler remembers that the variable `z` is defined as having the integer type. This information can then be re-used later when the compiler sees the same variable `z` in the program again and needs to check what the type of that variable is.

The same process then repeats itself at the next level further down the tree: using again the fourth semantic rule, the intype attribute is propagated one level down and at the same time the symbol table entry for `y` is updated by the addtype procedure to be the type integer:



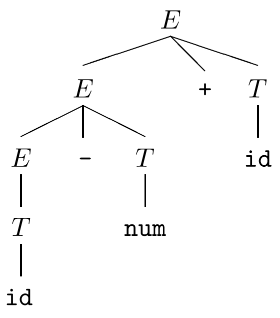
Finally the fifth semantic rule is used when we reach the bottom of the parse tree. Here there is no need to propagate anything further down, we simply use the addtype procedure again to update the type in the symbol entry for the identifier. The entry attribute of the `id` token is a pointer to the symbol table entry for the corresponding variable `x`:



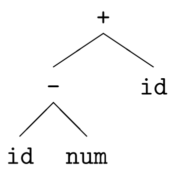
Note that computing a mix of synthesized and inherited attributes is more complicated than computing synthesized attributes only. If we have synthesized attributes only (i.e. S-attributes definitions) then any bottom-up order will work during the computation of the attributes. When we have inherited attributes then it is not always clear in which order the attributes should be computed to get the correct result. In the computation above, we started with computing the synthesized attribute syntype in the left part of the parse tree, then computed the inherited intype attribute in the right part of the parse tree. The reason we did it in that particular order is simply because we guessed that it would provide the correct answer! Unfortunately in general there is no simple way to guess an order in which to compute the attributes: one has to look at all the semantic rules, understand the dependencies that the rules create between the various attributes, decide on some computation order of the attributes based on their dependencies, and then convince oneself that that order is actually correct. . . Later we will see that in some cases it is nevertheless possible to use a fixed pre-established order even for inherited attributes, provided those inherited attributes fulfill some extra restrictions.

Syntax Trees

Here is now a third example of syntax-directed definitions. This example will again use only synthesized attributes (i.e. it will be S-attributed definitions) but here the semantic rules will be used mostly for their side effects. What we want to do is transform a parse tree into a syntax tree. For example, we want to transform a parse tree like this one:



into a syntax tree like this:



In a syntax tree all the nonterminals are gone and the interior nodes of the tree are the operators from the source program. This means that in a syntax tree tokens can be interior nodes, while in a parse tree only nonterminals can be interior nodes.

Syntax trees are nice because:

- they are much smaller than parse trees, so they use a lot less memory inside a compiler;
- since they do not contain nonterminals, the shape of a syntax tree only depends on the source program it represents, while the shape of a parse tree depends on both the source program it represents and the grammar that is used to parse that source program; another way to look at it is to say that a syntax tree is independent of the grammar which is used during the syntax analysis, while a parse tree is not.

That's why syntax trees are sometimes used inside compilers as a form of intermediate language (instead of using three-address code).

To construct a syntax tree starting from a parse tree, we are going to create exactly one syntax tree node for each token in the parse tree. Then we will manipulate pointers to these various nodes to connect them together to get the syntax tree. To this all this, we will use three procedures:

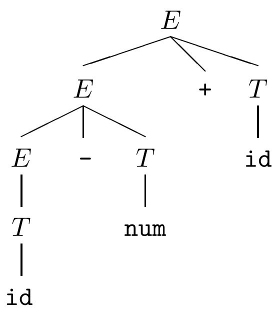
- mkleaf(`num`, `num`.val) that allocates a new leaf node for a `num` token and stores in it both the token and the corresponding value, which will come from the lexeme associated with the `num` token, since the val attribute is going to be intrinsic;
- mkleaf(`id`, `id`.entry) that allocates a new leaf node for an `id` token and stores in it both the token and the intrinsic entry attribute for the token, which is going to be a pointer to the corresponding entry in the the symbol table;
- mknode(`op`, *left*, *right*) that allocates a new interior node for the syntax tree and stores in it the token (called `op` here) for an operator, along with two pointers (called *left* and *right* here) pointing to the syntax tress for the two operands of the operator.

In addition to these three procedures and to the val and entry intrinsic attributes, we will also use a synthesized nprt ("Node PoinTeR") attribute to store pointers to the various syntax tree nodes that we will create using the three procedures above.

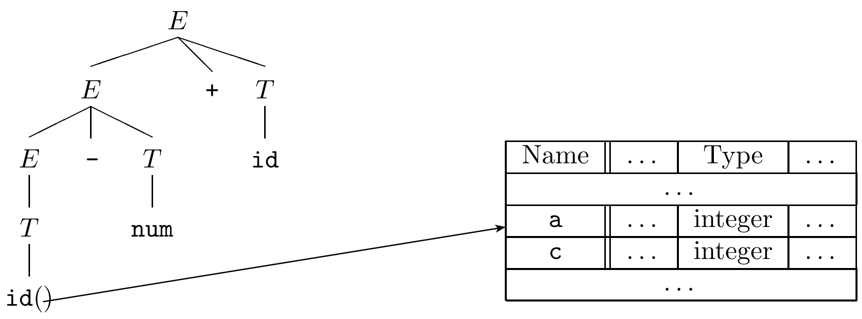
Given that all three attributes val, entry, and nprt are synthesized, the following syntax-directed definitions are then S-attributed definitions:

$E_1 \rightarrow E_2$ `+` T	E_1.nptr := mknode(`+`, E_2.nptr, T.nptr)
$E_1 \rightarrow E_2$ `-` T	E_1.nptr := mknode(`-`, E_2.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow$ `(` E `)`	T.nptr := E.nptr
$T \rightarrow$ `id`	T.nptr := mkleaf(`id`, `id`.entry)
$T \rightarrow$ `num`	T.nptr := mkleaf(`num`, `num`.val)

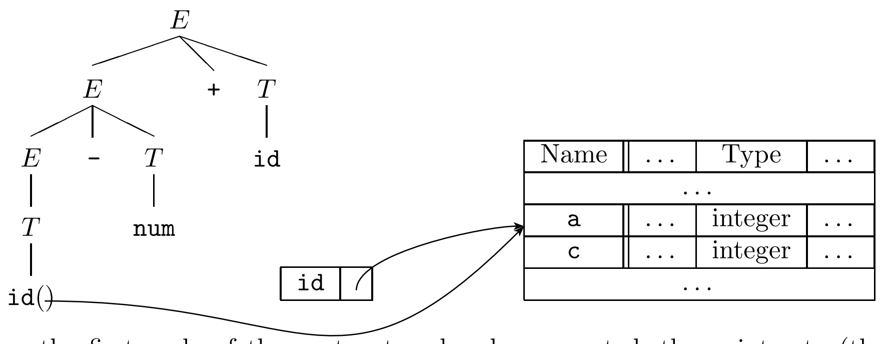
Let's apply these semantic rules to the following parse tree for the expression `a - 4 + c`:



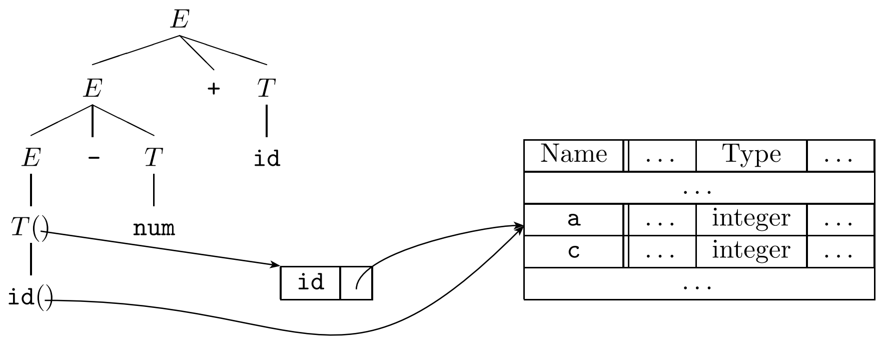
Since all the attributes in our semantic rules are synthesized. we can use any bottom-up order to compute them. Let's start for example with the leftmost identifier. First we show the value of `id`.entry which is an intrinsic attribute that points to the entry for the variable `a` in the symbol table:



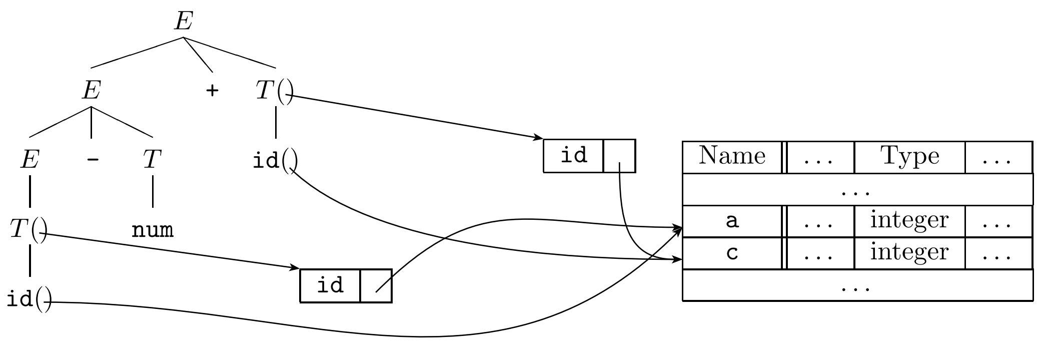
Once we have the value of `id`.entry we can use the semantic rule T.nptr := mkleaf(`id`, `id`.entry) to compute the nptr attribute of the T which is directly above the `id`. This rule uses the mkleaf procedure to create the first node in the syntax tree and put in that node both the token `id` and a copy of the pointer `id`.entry that points to the symbol table entry for `a`:



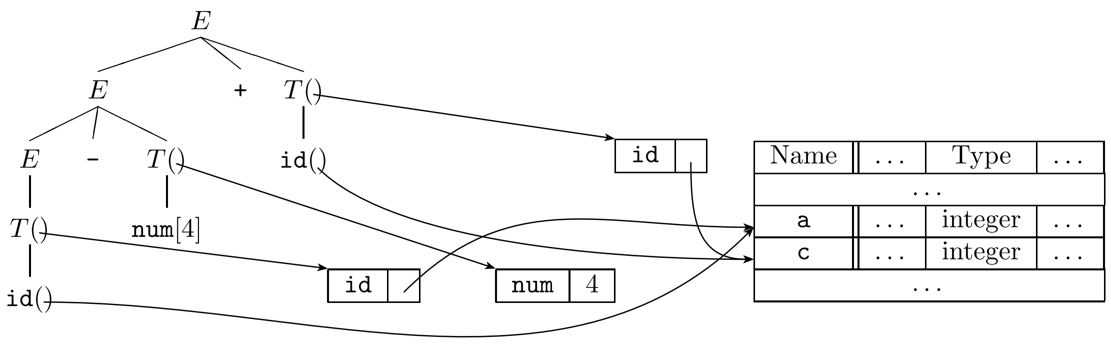
Once the first node of the syntax tree has been created, the pointer to (the memory address of) the node is assigned to the nptr attribute of T:



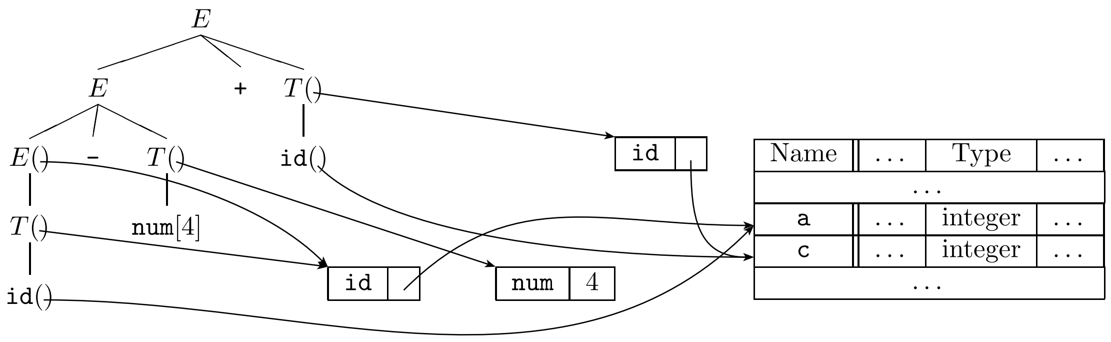
We can then do the same thing for the other identifier in the parse tree, which corresponds to the variable `c`:



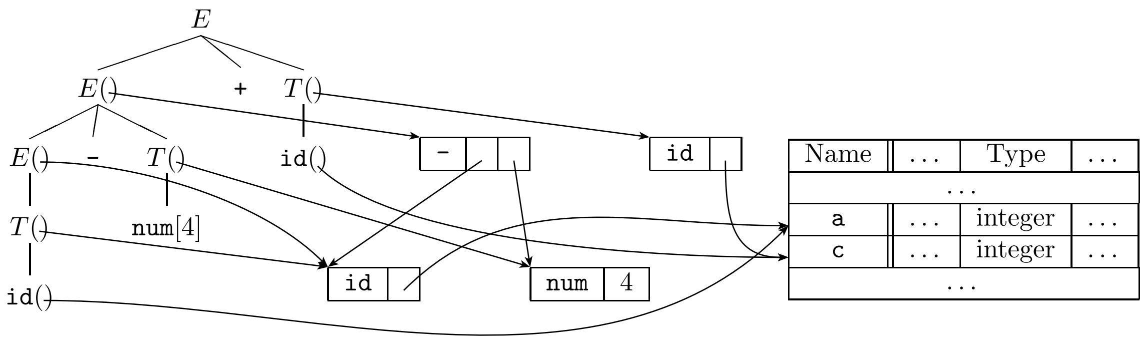
Using the semantic rule T.nptr := mkleaf(`num`, `num`.val) we do the same thing again, this time for the `num` token, except that, instead of storing in the new syntax tree node a pointer to an entry in the symbol table, we store in the node the value of the val attribute of the token `num`. Since that attribute val is intrinsic, we look at the lexeme corresponding to the token to determine that val should be 4 (which we show in the parse tree by putting it between brackets), then we use the mkleaf procedure to create the node, and assign the resulting node address to the nptr attribute of T:



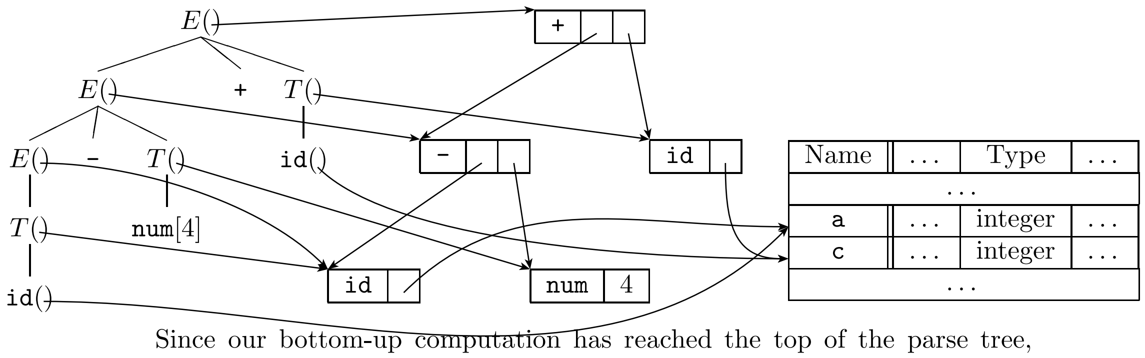
Next we just use the semantic rule E.nptr := T.nptr to compute the nptr attribute of the leftmost E by simply copying the nptr attribute of the leftmost T (which is a pointer to the first syntax node we create):



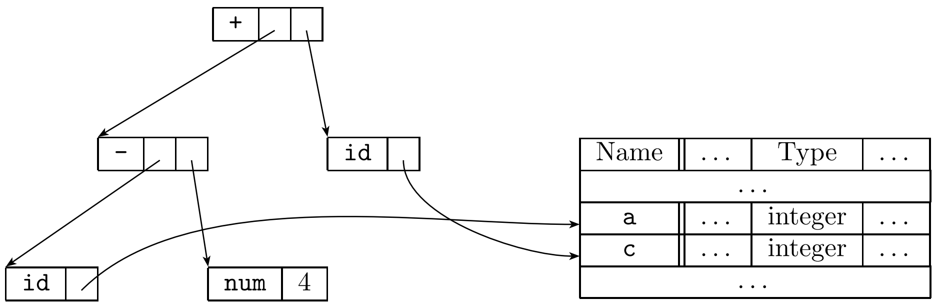
We now have to create the syntax tree node for the `-` operator. To do this we use the semantic rule E_1.nptr := mknode(`-`, E_2.nptr, T.nptr) associated with the grammar production for subtraction. Here the mknode function creates a new syntax tree node with three parts: the token `-`, one pointer to the syntax sub-tree for the left operand of the subtraction, and one pointer to the syntax sub-tree for the right operand of the subtraction. These two pointers come from the nptr attribute of the leftmost E (which points to the `id` syntax tree node for `a`) and from the nptr attribute of the middle T (which points to the `num` syntax tree node). So we just copy these two pointers into the new syntax tree node created by mknode, then assign the resulting pointer (the memory address at which mknode just allocated the new syntax node) to the nptr attribute of the middle E in the parse tree:



Finally we have to create the syntax tree node for the `+` operator. To do this we use the semantic rule E_1.nptr := mknode(`+`, E_2.nptr, T.nptr) associated with the grammar production for addition. Here again the mknode function creates a new syntax tree node with three parts: the token `+`, one pointer to the syntax sub-tree for the left operand of the subtraction, and one pointer to the syntax sub-tree for the right operand of the subtraction. This time the two pointers come from the nptr attribute of the middle E (which points to the `-` syntax tree node created at the previous step) and from the nptr attribute of the rightmost T (which points to the `id` syntax tree node for `c`). So we just copy these two pointers into the new syntax tree node created by mknode, then assign the resulting pointer (the memory address at which mknode just allocated the new syntax node) to the nptr attribute of the top E in the parse tree:



Since our bottom-up computation has reached the top of the parse tree, we know that the computation is over, and the nptr attribute of the top E in the parse tree points to the root of the corresponding syntax tree we have just created, which we show here without the parse tree next to it (but still with the symbol table):



You can compare the shape of this syntax tree with the shape of the syntax tree we first presented at the beginning of this section, and you will see that they have the same shape.

Evaluation of S-attributed Definitions

As we have indicated above, in S-attributed definitions all the attributes are synthesize and therefore the computation of the attributes can always be done in any bottom-up order. In this section we look at how a compiler actually does this. The idea is in fact very simple and we already almost entirely described it in the last section of the previous lecture notes on bottom-up parsing.

Remember that when a bottom-up parser analyzes some input, it constructs the parse tree bottom-up (hence the name for that kind of parsers...) Since the attributes in S-attributed definitions can be computed in any bottom-up order, we know that using the same bottom-up order as the parser does will always gives us the right result. This means that we can in fact compute all the synthesized attributes at the *same time* as we construct the parse tree itself. In other words, if we only have synthesized attributes, then it is not necessary to have one compiler phase for the syntax analysis and another separate compiler phase for the semantic analysis (which in this course we will always do using attributes), we can just do both syntax and semantic analysis at the same time and compute all our attributes at the same time as we are building the parse tree.

We do this just like we described in the last section of the previous lecture notes: we extend the bottom-up parser's stack to also contain a semantic value associated with each grammar symbol on the stack (i.e. use triples on the parser's stack instead of pairs):

- if the parser shifts a token from the input onto the stack, then we know that the associated attribute is intrinsic (a synthesized attribute for a token is always intrinsic) and therefore we get the value of the attribute from somewhere else (the corresponding lexeme, or the corresponding entry in the symbol table, etc.);

- if the parser reduces the top of the stack, then we use the semantic rule associated with the grammar production used for the reduction to compute the attribute of the new nonterminal at the top of the stack.

In short, using a bottom-up parser makes is extremely easy to compute synthesized attributes. For example, if you use an LALR(1) parser generated by YACC, then the only thing there is to do to implement a semantic rule like:

$$E_1.\mathrm{val} := E_2.\mathrm{val} + T.\mathrm{val}$$

is to rewrite it like:

```
$$ = $1 + $3;
```

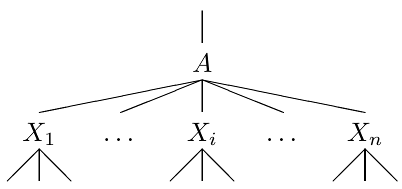
and write this C action on the same line as the corresponding grammar production in the the YACC specification file for your grammar. Here `$$` always represents the attribute of the LHS of the grammar production, and `$1`, `$2`, `$3`, etc, represent the attributes of the various grammar symbols in the RHS of the grammar production. The values of the intrinsic attributes of tokens are simply copied from the `yylval` global variable which is shared with the lexer.

Evaluation of L-attributed Definitions

What we have not explained yet is how to evaluate inherited attributes. Since for inherited attributes the computation might have to go top-down, or left-right, or right-left, it is not possible in the general case to compute these attributes at the same time as the parse tree is constructed by the parser. It might then be necessary to have in the compiler a separate semantic analysis phase which is implemented by hand (not generated automatically by a tool) and which uses a specific order to compute the different attributes of the grammar symbols in the parse tree. If the semantic rules are changed then the compiler implementor might have to find and implement a new specific execution order for the new semantic rules to ensure that all the attributes are computed in the right order. This explains why it is usually a good idea to avoid using inherited attributes as much as possible, as computing them can be very painful.

There is one case though in which computing inherited attributes it still manageable. This is when all the attributes can be computed during a single depth-first traversal of the parse tree. During such a traversal, inherited attributes are evaluated when moving down in the parse tree and synthesized attributes are evaluated when moving up in the parse tree.

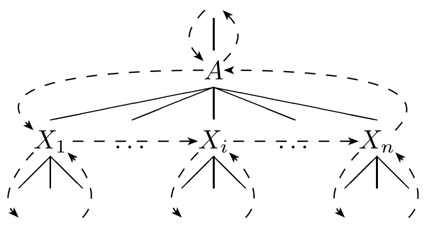
If part of the parse tree looks like this:



then a depth-first traversal of this piece of parse tree works like this:

```
df-traversal(node $A$){
    evaluate the inherited attributes of $A$
    for each child $X_i$ of $A$ (in left-to-right order), do{
        df-traversal($X_i$)
    }
    evaluate the synthesized attributes of $A$
}
```

In other words, when using such a depth-first traversal, the attributes are evaluated in the following order:



If all the synthesized and inherited attributes of some syntax-directed definitions can be evaluated using such a depth-first traversal of the parse tree, then the syntax-directed definitions are called *L-attributed definitions*. In other words, L-attributed definitions are syntax-directed definitions for which a depth-first evaluation of the attributes is guaranteed to work.

Having L-attributed definitions means that the computation of inherited attributes can happen top-down and left-right, as before, but cannot be done right-left anymore. This means that, for L-attributed definitions, the shape of the semantic rules is more restricted. Recall that, in the general case of syntax-directed definitions, the shape of a semantic rule for an inherited attribute is:

$$X_i.x_i := f(A.a, X_1.x_1, \ldots, X_{i-1}.x_{i-1}, X_{i+1}.x_{i+1}, \ldots, X_n.x_n)$$

In the more restricted case of L-attributed definitions though, the shape of a semantic rule for an inherited attribute is limited to be:

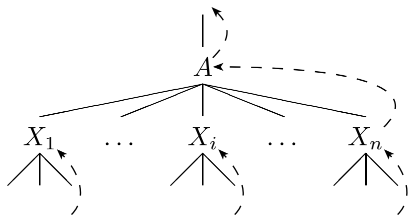
$$X_i.x_i := f(A.a, X_1.x_1, \ldots, X_{i-1}.x_{i-1})$$

In other words, the inherited attribute $X_i.x_i$ can depend on the attribute $A.a$ of the parent A of X_i and on the attributes $X_1.x_1, \ldots, X_{i-1}.x_{i-1}$ of the siblings X_1,

…, X_{i-1} on the left of X_i, but it cannot depend on the attributes $X_{i+1}.x_{i+1}$, …, $X_n.x_n$ of the siblings X_{i+1}, …, X_n on the right of X_i.

If you look back at the second example of syntax-directed definitions that we gave above, the one that uses a combination of a syntype synthesized attribute and of a intype inherited attribute to compute the types of variables in variable definitions, you will see that we used a depth-first order when computing the attributes of the parse tree for the variable definitions `int x, y, z`. This works because the semantic rules that we wrote for the inherited attribute intype all follow the more restricted shape described just above. In turn it means that the semantic rules we wrote for the intype and syntype attributes are in fact L-attributed definitions.

Note that all S-attributed definitions are also L-attributed definitions. This is easy to see if you think of S-attributed definitions as just being L-attributed definitions that do not use any inherited attributes (i.e. all the attributes are synthesized). Since S-attributed definitions can be computed in any bottom-up order, they can in particular be computed in a left-to-right bottom-up order that matches a depth-first order: simply do a depth-first traversal just like for L-attributed definitions and compute the synthesized attributes of the S-attributed definitions only when your depth-first traversal is moving up:



So S-attributed definitions are just a special case of L-attributed definitions.

Since S-attributed definitions can be implemented using a bottom-up parser, an interesting question is then whether L-attributed definitions can also be implemented using a bottom-up parser or not. The answer is: it depends. If the associated grammar is simple enough to be LL(1) then, by suitably modifying the grammar productions and the corresponding semantic rules of the L-attributed definitions, it is always possible to correctly compute all the attributes of the L-attributed definitions using a bottom-up parser (see the Dragon Book for the details of how to suitably modify the grammar productions and semantic rules). This is not an obvious result, since it means that in some cases you can compute inherited attributes, which might normally require a top-down order of computation, using a bottom-up parser!

If the grammar associated with the L-attributed definitions is LR(1) instead of being LL(1), then the same result does *not* hold: in some cases it will be possible to use a bottom-up parser to compute the inherited attributes, but in some other cases it will not be possible. This means that, for inherited attributes, if your semantic rules are simple enough to be L-attributed definitions but your grammar is complicated enough to not be LL(1), then using a bottom-up parser to compute the attributes might no work. Since most general-purpose programming languages are not simple enough to be LL(1) (Pascal is one well known exception) but are in fact LR(1), then, if you have semantic rules that require using inherited attributes, trying to compute these attributes at the same time as the parse tree is constructed by the bottom-up parser might or might not work. You will then have to look at your semantic rules, think hard, and convince yourself that your semantic rules do indeed work with your bottom-up parser. This is in particular the case when you have inherited attributes and you use an automated tool like YACC to generate the bottom-up parser. If you cannot convince yourself that the computation order used by the YACC-generated parser works with your semantic rules (which means you have to first understand in details the order in which the bottom-up parser does its shifts and reduces...) then you will have instead to implement your semantic rules in a separate semantic phase in your compiler, and decide yourself on a computation order which you know will work with your semantic rules. This will be of course a lot more work than simply writing semantic rules as part of a YACC specification file, which again explains why avoiding the use of inherited attributes is in general a good idea...