بسم اللّٰه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

L-Attributed Definitions

The second class of SDD's is called L-attributed definitions. The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence "L-attributed").

More precisely, each attribute must be either

1. *Synthesized, or*
2. *Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X_1X_2 \dots X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:*
   - *(a) Inherited attributes associated with the head A.*
   - *(b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, X_2, \dots, X_{i-1}$ located to the left of X_i.*
   - *(c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.*

Example 5.8 : The SDD in Fig. 5.4 is L-attributed. To see why, consider the semantic rules for inherited attributes, which are repeated here for convenience:

PRODUCTION	SEMANTIC RULE
$T \to F\,T'$	$T'.inh = F.val$
$T' \to *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$

The first of these rules defines the inherited attribute $T'.inh$ using only $F.val$, and F appears to the left of T' in the production body, as required. The second rule defines $T'_1.inh$ using the inherited attribute $T'.inh$ associated with the head, and $F.val$, where F appears to the left of T'_1 in the production body.

In each of these cases, the rules use information "from above or from the left," as required by the class. The remaining attributes are synthesized. Hence, the SDD is L-attributed. □

Example 5.9 : Any SDD containing the following production and rules cannot be L-attributed:

PRODUCTION	SEMANTIC RULES
$A \to B\ C$	$A.s = B.b;$
	$B.i = f(C.c, A.s)$

The first rule, $A.s = B.b$, is a legitimate rule in either an S-attributed or L-attributed SDD. It defines a synthesized attribute $A.s$ in terms of an attribute at a child (that is, a symbol within the production body).

The second rule defines an inherited attribute $B.i$, so the entire SDD cannot be S-attributed. Further, although the rule is legal, the SDD cannot be L-attributed, because the attribute $C.c$ is used to help define $B.i$, and C is to the right of B in the production body. While attributes at siblings in a parse tree may be used in L-attributed SDD's, they must be to the left of the symbol whose attribute is being defined. □

In an L-attributed grammar, attributes can be evaluated by visiting the nodes of the parse tree in a single left-to-right, depth-first traversal (the same order in which they are visited during a top-down parse). If we say that an attribute $A.s$ depends on an attribute $B.t$ if $B.t$ is ever passed to a semantic rule that returns a value for $A.s$, then we can define L-attributed grammars more formally with the following two rules:

(1) each synthesized attribute of a left-hand-side symbol depends only on that symbol's own inherited attributes or on attributes (synthesized or inherited) of the production's right-hand-side symbols, and

(2) each inherited attribute of a right-hand-side symbol depends only on inherited attributes of the left-hand-side symbol or on attributes (synthesized or inherited) of symbols to its left in the right-hand side.

ارتباط میان گرامرهای ***L-attributed*** و ***S-attributed***

Because L-attributed grammars permit rules that initialize attributes of the left-hand side of a production using attributes of symbols on the right-hand side, every S-attributed grammar is also an L-attributed grammar. The reverse is not the case: S-attributed grammars do not permit the initialization of attributes on the right-hand side, so there are L-attributed grammars that are not S-attributed.

Just as context-free grammars can be categorized according to the parsing algorithm(s) that can use them, attribute grammars can be categorized according to the complexity of their pattern of attribute flow.

☞ *S-attributed grammars, in which all attributes are synthesized, can naturally be evaluated in a single bottom-up pass over a parse tree, in precisely the order the tree is discovered by an LR-family parser.*

☞ *L-attributed grammars, in which all attribute flow is depth-first left-to-right, can be evaluated in precisely the order that the parse tree is predicted and matched by an LL-family parser.*

Attribute grammars with more complex patterns of attribute flow are not commonly used for the parse trees of production compilers, but are valuable for syntax-based editors, incremental compilers, and various other tools.

دربارهٔ ادغام تحلیل معنایی با فرایند تجزیه

Because attribute flow in an S-attributed grammar is strictly bottom-up, attributes can be evaluated by visiting the nodes of the parse tree in exactly the same order that those nodes are generated by an LR-family parser. In fact, the attributes can be evaluated on the fly during a bottom-up parse, thereby interleaving parsing and semantic analysis (attribute evaluation).

دربارهٔ ادغام تحلیل معنایی با فرایند تجزیه

☞ *S-attributed attribute grammars are the most general class of attribute grammars for which evaluation can be implemented on the fly during an LR parse.*

☞ *L-attributed grammars are the most general class for which evaluation can be implemented on the fly during an LL parse.*

If we interleave semantic analysis (and possibly intermediate code generation) with parsing, then a bottom-up parser must in general be paired with an S-attributed translation scheme; a top-down parser must be paired with an L-attributed translation scheme.

Semantic Rules with Controlled Side Effects

In practice, translations involve side effects: a desk calculator might print a result; a code generator might enter the type of an identifier into a symbol table. With SDD's, we strike a balance between attribute grammars and translation schemes. Attribute grammars have no side effects and allow any evaluation order consistent with the dependency graph. Translation schemes impose left-to-right evaluation and allow semantic actions to contain any program fragment; translation schemes are discussed in Section 5.4.

We shall control side effects in SDD's in one of the following ways:

☞ *Permit incidental side effects that do not constrain attribute evaluation. In other words, permit side effects when attribute evaluation based on any topological sort of the dependency graph produces a "correct" translation, where "correct" depends on the application.*

☞ *Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order. The constraints can be thought of as implicit edges added to the dependency graph.*

As an example of an incidental side effect, let us modify the desk calculator of Example 5.1 to print a result. Instead of the rule $L.val = E.val$, which saves the result in the synthesized attribute $L.val$, consider:

	PRODUCTION	SEMANTIC RULE
1)	$L \to E\ \mathbf{n}$	$print(E.val)$

Semantic rules that are executed for their side effects, such as $print(E.val)$, will be treated as the definitions of dummy synthesized attributes associated with the head of the production. The modified SDD produces the same translation under any topological sort, since the print statement is executed at the end, after the result is computed into $E.val$.

Example 5.10: The SDD in Fig. 5.8 takes a simple declaration D consisting of a basic type T followed by a list L of identifiers. T can be **int** or **float**. For each identifier on the list, the type is entered into the symbol-table entry for the identifier. We assume that entering the type for one identifier does not affect the symbol-table entry for any other identifier. Thus, entries can be updated in any order. This SDD does not check whether an identifier is declared more than once; it can be modified to do so.

	PRODUCTION	SEMANTIC RULES
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \mathbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \mathbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \mathbf{id}$	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5)	$L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

Figure 5.8: Syntax-directed definition for simple type declarations

Nonterminal D represents a declaration, which, from production 1, consists of a type T followed by a list L of identifiers. T has one attribute, $T.type$, which is the type in the declaration D. Nonterminal L also has one attribute, which we call inh to emphasize that it is an inherited attribute. The purpose of $L.inh$ is to pass the declared type down the list of identifiers, so that it can be added to the appropriate symbol-table entries.

Productions 2 and 3 each evaluate the synthesized attribute $T.type$, giving it the appropriate value, integer or float. This type is passed to the attribute $L.inh$ in the rule for production 1. Production 4 passes $L.inh$ down the parse tree. That is, the value $L_1.inh$ is computed at a parse-tree node by copying the value of $L.inh$ from the parent of that node; the parent corresponds to the head of the production.

Productions 4 and 5 also have a rule in which a function $addType$ is called with two arguments:
1. $id.entry$, a lexical value that points to a symbol-table object, and
2. $L.inh$, the type being assigned to every identifier on the list.
We suppose that function $addType$ properly installs the type $L.inh$ as the type of the represented identifier.

A dependency graph for the input string* float $\mathbf{id}_1, \mathbf{id}_2, \mathbf{id}_3$ *appears in Fig. 5.9. Numbers 1 through 10 represent the nodes of the dependency graph. Nodes 1, 2, and 3 represent the attribute* $entry$ *associated with each of the leaves labeled* id. *Nodes 6, 8, and 10 are the dummy attributes that represent the application of the function* $addType$ *to a type and one of these* $entry$ *values.

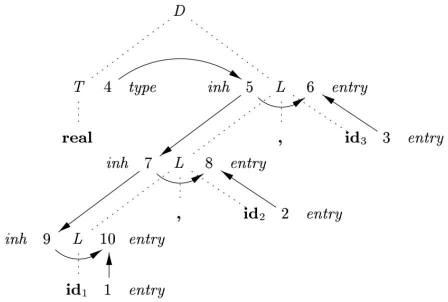


Figure 5.9: Dependency graph for a declaration **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$

Node 4 represents the attribute $T.type$, and is actually where attribute evaluation begins. This type is then passed to nodes 5, 7, and 9 representing $L.inh$ associated with each of the occurrences of the nonterminal L.

Intermediate-Code Generation

مثلاً فصل ششم از کتاب آهو و فصل هفتم از کتاب اپل را ببینید.

یک کامپایلر متشکل از دو بخش عمده است: آنالیز و سنتز

☞ *The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.*

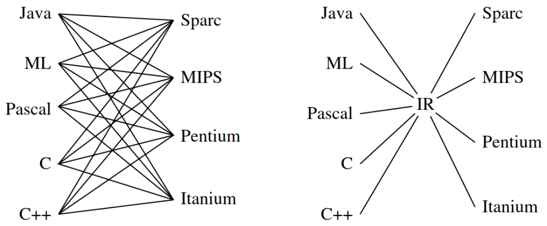
☞ *The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end.*

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. Ideally, details of the source language are confined to the front end, and details of the target machine to the back end. With a suitably defined intermediate representation, a compiler for language i and machine j can then be built by combining the front end for language i with the back end for machine j. This approach to creating suite of compilers can save a considerable amount of effort: $m \times n$ compilers can be built by writing just m front ends and n back ends.

The semantic analysis phase of a compiler must translate abstract syntax into abstract machine code. It can do this after type-checking, or at the same time. Though it is possible to translate directly to real machine code, this hinders portability and modularity. Suppose we want compilers for N different source languages, targeted to M different machines. In principle this is $N \times M$ compilers, a large implementation task. An intermediate representation (IR) is a kind of abstract machine language that can express the target-machine operations without committing to too much machine-specific detail. But it is also independent of the details of the source language.

The front end of the compiler does lexical analysis, parsing, semantic analysis, and translation to intermediate representation. The back end does optimization of the intermediate representation and translation to machine language.

A portable compiler translates the source language into IR and then translates the IR into machine language. Now only N front ends and M back ends are required. Such an implementation task is more reasonable. Even when only one front end and one back end are being built, a good IR can modularize the task, so that the front end is not complicated with machine-specific details, and the back end is not bothered with information specific to one source language. Many different kinds of IR are used in compilers.



Compilers for five languages and four target machines:
(a) without an IR, (b) with an IR.

☞ *The front end of a compiler constructs an intermediate representation of the source program from which the back end generates the target program.*

☞ *The intermediate representation (IR) forms an interface between specific compiler phases, typically between the front-end and the back-end of the compiler. The last phase of the front-end is usually the semantic analyzer and the back-end is responsible for code generation. The IR should be designed to facilitate the interface between the front-end and the back-end.*

☞ *There is no reason why a compiler cannot be designed to use more than one IR.*

☞ *The IR should be sufficiently expressive so that all the constructs of the source language can be encoded cleanly into IR statements or structures. Assuming that we are thinking about a traditional compiler generating code for a target machine, the IR should be easily translatable into target machine code.*

☞ *Another important characteristic of an IR is that it should be able to support aggressive code optimization, thus permitting the generation of high-quality code. In many modern compilers this is the IR's key design aim.*

☞ *After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.*

Two Kinds of Intermediate Representations:

The two most important intermediate representations are:

☞ *Trees, including parse trees and (abstract) syntax trees.*

☞ *Linear representations, especially "three-address code."*

Abstract and Concrete Syntax

In an abstract syntax tree for an expression, each interior node represents an operator; the children of the node represent the operands of the operator.

More generally, any programming construct can be handled by making up an operator for the construct and treating as operands the semantically meaningful components of that construct.

Abstract syntax trees (ASTs), or simply syntax trees, resemble parse trees to an extent. However, in the syntax tree, interior nodes represent programming constructs while in the parse tree, the interior nodes represent nonterminals.

In the abstract syntax tree for $9 - 5 + 2$ in Fig. 2.22, the root represents the operator $+$. The subtrees of the root represent the subexpressions $9 - 5$ and 2. The grouping of $9 - 5$ as an operand reflects the left-to-right evaluation of operators at the same precedence level. Since $-$ and $+$ have the same precedence, $9 - 5 + 2$ is equivalent to $(9 - 5) + 2$.

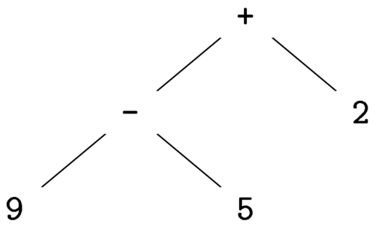


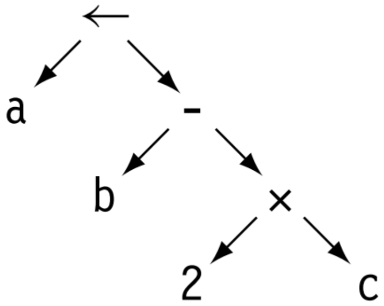
Figure 2.22: Syntax tree for 9-5+2

☞ *Many nonterminals of a grammar represent programming constructs, but others are "helpers" of one sort of another, such as those representing terms, factors, or other variations of expressions. In the syntax tree, these helpers typically are not needed and are hence dropped. To emphasize the contrast, a parse tree is sometimes called a concrete syntax tree, and the underlying grammar is called a concrete syntax for the language.*

☞ *In the syntax tree in Fig. 2.22, each interior node is associated with an operator, with no "helper" nodes for single productions (a production whose body consists of a single nonterminal, and nothing else) like $expr \to term$ or for ε-productions like $rest \to \varepsilon$.*

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. ASTs are a form of intermediate representation.

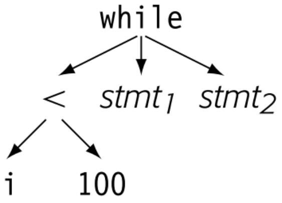
چند مثال از *AST*ها



AST for $a \leftarrow b - 2 \times c$

```
while (i < 100)
    begin
        stmt1
        end
stmt2
```

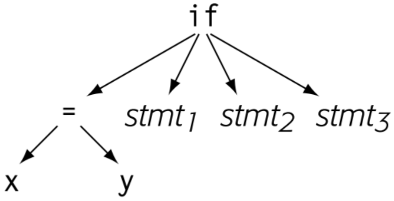
Source Code



Abstract Syntax Tree

```
if (x = y)
   then stmt1
   else stmt2
stmt3
```

Source Code



Abstract Syntax Tree

Not every detail of the sequence of reductions performed by the parser is reflected in the tree. Nevertheless, the data in the tree is sufficient for later stages of compilation.