

# **Programming Language Concepts and Semantics Part I**

## **Lecture 4 Translational Semantics**

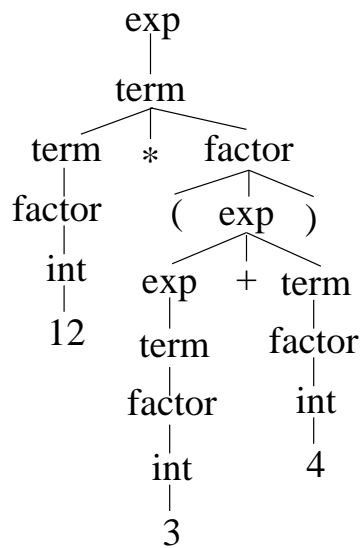
Lecture IV will continue to discuss parsing and explain syntax-directed translation:

- Attribute Grammars
- DCGs with Attributes
- Abstract Syntax and Parse Trees
- Context Free Grammars
- Syntax-directed Translation / Translational Semantics

## Syntax-directed Translation (Idea)

The Idea of syntax-directed translation is to first build an abstract representation of the programs structure (a syntax tree) and then, in a second phase, to build the translation of the program by traversing this tree according to syntactic structure and recursively collecting a complete translation from local translations.

**Example:** Translation from Infix to Postfix



*How do you traverse this tree to build a postfix expression?*

## Attribute Grammars

To be able to build a structural representation, such as the parse tree, explicitly we need a mechanism to construct, manipulate and pass data structures during parsing.

In this way we can

- build an explicit representation of the syntax tree or
- interleave translation or execution with the parsing

This idea is embodied in **Attribute Grammars** (Knuth, 1968), which extend normal grammars by adding attributes to non-terminals.

Attribute Grammars can also be used to steer production applications with additional tests.

There are two attribute types:

- *inherited attributes* are provided when a production is called (by the caller, i.e. another production).
- *synthesized attributes* are composed by a production and tested or used to compute synthesized attributes.

## Attribute Grammars (cont.)

With each grammar production  $A \rightarrow \alpha$  we associate a set of semantic rules of the form

$$b := f(c_1, \dots, c_n)$$

where  $f$  is a (usually side effect-free) function,  $c_1, \dots, c_n$  are attributes of the symbols in the rule and either:

- $b$  is an attribute of  $A$ , in which case  $b$  is a **synthesized** attribute.
- $b$  is an attribute of one of the symbols in  $\alpha$ , in which case  $b$  is an **inherited** attribute.

More generally, a production could also specify

- **conditions** on attribute values for a production to be applicable,
- **procedures** to be evaluated which, for example, might update the symbol table.

A parse tree showing the values of the attributes at each node is said to be **annotated** or **decorated**.

## Computation of Attributes

- Inherited attributes in a production  $P : X \rightarrow X_1, \dots, X_n$  for a non-terminal  $X_i$  are computed from
  1. inherited attributes of  $X$ ,
  2. synthesized attributes on the right-hand side in  $X_1, \dots, X_{i-1}$ ,
  3. attributes of terminals on the right-hand side in  $X_1, \dots, X_{i-1}$ .
- Synthesized attributes in a production  $P : X \rightarrow X_1, \dots, X_n$  for a non-terminal  $X$  are computed from
  1. inherited attributes of  $X$ ,
  2. synthesized attributes on the right-hand side of  $P$ ,
  3. attributes of terminals on the right-hand side of  $P$ .

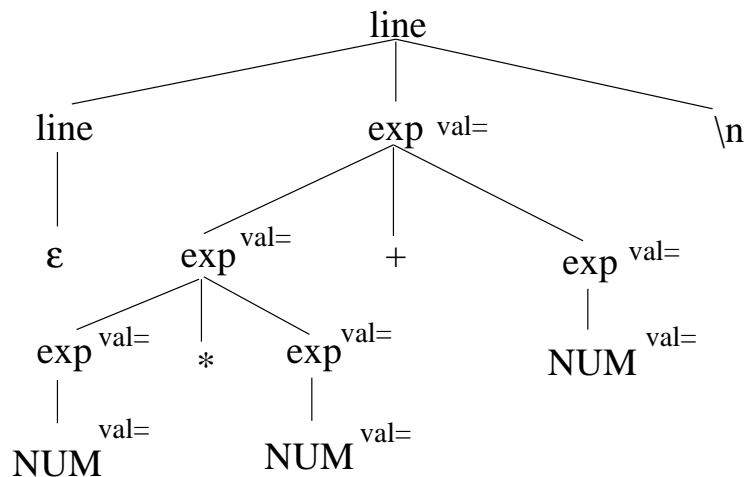
## Attribute Grammar – Synthesized

A simple example of an attribute-grammar that uses only synthesized attributes is:

Production	Semantic Rules
$line \rightarrow \epsilon$	
$line \rightarrow line\ exp\ \backslash n$	$print(exp.val)$
$exp_0 \rightarrow exp_1 + exp_2$	$exp_0.val := exp_1.val + exp_2.val$
$exp_0 \rightarrow exp_1 - exp_2$	$exp_0.val := exp_1.val - exp_2.val$
$exp_0 \rightarrow exp_1 * exp_2$	$exp_0.val := exp_1.val * exp_2.val$
$exp_0 \rightarrow exp_1 / exp_2$	$exp_0.val := exp_1.val / exp_2.val$
$exp_0 \rightarrow (exp_1)$	$exp_0.val := exp_1.val$
$exp_0 \rightarrow NUM$	$exp_0.val := NUM.val$

In this grammar  $exp$  and  $NUM$  have a single synthesized attribute  $val$  and  $line$  and the remaining terminal symbols have no attributes.

The annotated parse tree for  $3 * 5 + 4 \backslash n$  is:



## Attributed DCGs / Synthesized Attributes

DCGs can also use Attributes. They are given as arguments to the non-terminals in the same way that predicate functors are given arguments. If additional computations or tests are required, they are written as normal Prolog subgoals on the RHS and enclosed in curly brackets (see below).

The equivalent DCG production for

$$exp_0 \rightarrow (exp_1) [ exp_0.val := exp_1.val ]$$

is

$$exp(V) \text{ --> } ['('], exp(V), [')'].$$

In the same way we could translate

$$exp_0 \rightarrow exp_1/exp_2 [ exp_0.val := exp_1.val/exp_2.val ]$$

into

$$exp(V) \text{ --> } exp(V1), ['/'], exp(V2), \{ V \text{ is } V1 / V2 \}.$$

*Unfortunately, this rule won't work. Why?*

We will see a working DCG version of the expression grammar a bit later.

## Attribute Grammar – Inherited

A simple example of an attribute-grammar that uses only inherited attributes is:

Production	Semantic Rules
$exp_1 \rightarrow term + exp_2$	$term.rep := exp_1.rep;$ $exp_2.rep := exp_1.rep;$
$exp \rightarrow term$	$term.rep := exp.rep$
$term \rightarrow NUM$	$NUM.rep := term.rep$
$NUM \rightarrow ZERO \mid ONE \mid \dots \mid NINE$	if $NUM.rep = 'words'$
$NUM \rightarrow 0 \mid 1 \mid \dots \mid 9$	if $NUM.rep = 'digits'$

In this grammar  $term$ ,  $exp$  and  $NUM$  have a single inherited attribute  $rep$  that switches between two types of representations in the terminals.

## Attributed DCGs / Inherited Attributes

The same grammar as a DCG is:

```
exp(Rep) --> term(Rep), exp(Rep).
```

```
exp(Rep) --> term(Rep).
```

```
term(Rep) -> num(Rep).
```

```
num('words') --> ['zero'].
```

```
.
```

```
.
```

```
.
```

```
num('words') --> ['nine'].
```

```
num('digits') --> ['0'].
```

```
.
```

```
.
```

```
.
```

```
num('digits') --> ['9'].
```

## Attribute Grammar – Mixed Attributes

In particular grammars that have been left-factored or DCGs that have been designed not to use backtracking (formally: have been made LL(k)) often lose the “intuitive” structure of the grammars, so that even simple computations require a mix of inherited and synthesized attributes.

Consider our grammar for arithmetic expressions.

$$\begin{aligned} \text{exp} &\rightarrow \text{term exp}' && \text{exp}' \cdot v1 := \text{term} \cdot v; \text{exp} \cdot v := \text{exp}' \cdot v; \\ \text{exp}' &\rightarrow +\text{exp} && \text{exp}' \cdot v := \text{exp}' \cdot v1 + \text{exp} \cdot v; \\ \text{exp}' &\rightarrow \epsilon && \text{exp}' \cdot v := \text{exp}' \cdot v1; \\ \text{term} &\rightarrow \text{factor term}' && \text{term}' \cdot v1 := \text{factor} \cdot v; \text{term} \cdot v := \text{term}' \cdot v; \\ \text{term}' &\rightarrow * \text{term} && \text{term}' \cdot v := \text{term}' \cdot v1 * \text{term} \cdot v; \\ \text{term}' &\rightarrow \epsilon && \text{term}' \cdot v := \text{term}' \cdot v1; \\ \text{factor} &\rightarrow \mathbf{int} && \text{factor} \cdot v := \mathbf{int} \cdot v; \\ \text{factor} &\rightarrow (\text{exp}) && \text{factor} \cdot v := \text{exp} \cdot v; \end{aligned}$$

## Attributed DCG with Mixed Attributes

The DCG equivalent of the grammar above is:

`exp(V) --> term(V1), exp1(V1,V).`

`exp1(V1,V) --> ['+'], exp(V2), { V is V1 + V2 }.`

`exp1(V1,V) --> ['-'], exp(V2), { V is V1 - V2 }.`

`exp1(V,V) --> [].`

`term(V) --> factor(V1), term1(V1,V).`

`term1(V1,V) --> ['*'], term(V2), { V is V1 * V2 }.`

`term1(V1,V) --> ['/'], term(V2), { V is V1 / V2 }.`

`term1(V,V) --> [].`

`factor(X) --> [X], { number(X) }.`

`factor(V) --> ['('], exp(V), [')'].`

## Well-Defined Attribute Grammars

An attribute grammar is **well-defined** if every attribute is defined and for no sentence does the dependency graph contain a cycle. (ie it must be a **dag**-directed acyclic graph).

For every dag, we can list the attributes so that the attribute comes after those attributes on which it depends. (Using **topological sorting**).

For a DCG the corresponding program must be able to evaluate the attributes. As Prolog uses unification and backtracking, this is not a concern in a “clean” DCG, but it becomes a concern if the DCG involves Prolog features that are outside the “clean” logical core, such as arithmetic tests (these require that all arguments to an expression are bound).

## Expressive Power of Attribute Grammars

Earlier we have pointed out that attribute grammars have higher expressive power than normal context-free grammars.

Example:

We know that  $L = a^n b^m c^n d^m$  is not a context-free language.

It is, of course, easy to write an attribute grammar for  $L$ :

Production	Semantic Rules
$s \rightarrow as\ bs\ cs\ ds$	if $c.n = a.n \wedge d.n := b.n$
$as \rightarrow \epsilon$	$as.n := 0$
$as_1 \rightarrow Aas_2$	$as_1.n := as_2.n + 1;$
$bs \rightarrow \epsilon$	$bs.n := 0$
$bs_1 \rightarrow Bbs_2$	$bs_1.n := bs_2.n + 1;$
$cs \rightarrow \epsilon$	$cs.n := 0$
$cs_1 \rightarrow Ccs_2$	$cs_1.n := cs_2.n + 1;$
$ds \rightarrow \epsilon$	$ds.n := 0$
$ds_1 \rightarrow Dds_2$	$ds_1.n := ds_2.n + 1;$

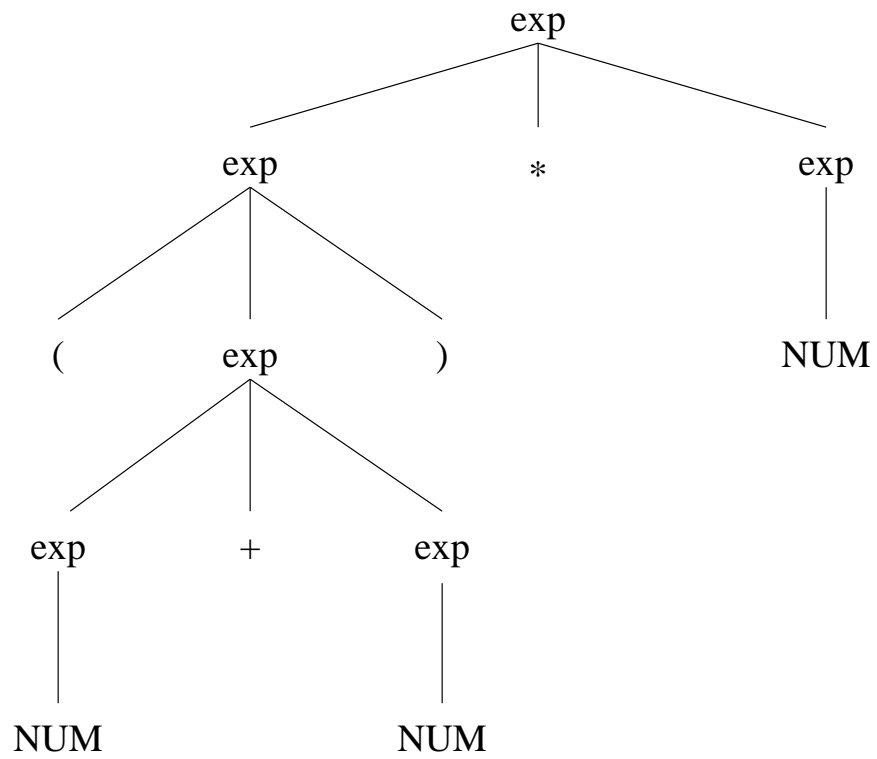
## Construction of Structure Trees

Earlier we noted that usually the parser does not build the full parse tree but rather strips it to the essential **structure tree**, (sometimes called an **abstract syntax tree**). This process can be specified using attributes.

Production	Semantic Rules
$exp_0 \rightarrow exp_1 + exp_2$	$exp_0.tree := tree('+', exp_1.tree, exp_2.tree)$
$exp_0 \rightarrow exp_1 - exp_2$	$exp_0.tree := tree('-', exp_1.tree, exp_2.tree)$
$exp_0 \rightarrow exp_1 * exp_2$	$exp_0.tree := tree('*', exp_1.tree, exp_2.tree)$
$exp_0 \rightarrow exp_1 / exp_2$	$exp_0.tree := tree('/', exp_1.tree, exp_2.tree)$
$exp_0 \rightarrow (exp_1)$	$exp_0.tree := exp_1.tree$
$exp_0 \rightarrow NUM$	$exp_0.tree := leaf(NUM.val)$

The function *leaf* is the type constructor for a leaf node and *tree* is the constructor for non-leaf nodes.

For example, evaluation of the parse tree  $(2 + 4) * 3$  leads to the abstract syntax tree:



## Computing Structure Trees with DCGs

We can easily modify our first DCG to produce the structure tree of the expression instead of computing the value. The structure tree will be represented in Prolog as a term where the main functor is the node type and the arguments are the subtrees.

```
exp(V)                                --> term(V1), exp1(V1,V).

exp1(V1, tree('+',V1,V2))             --> ['+'], exp(V2).
exp1(V1, tree('-',V1,V2))             --> ['-'], exp(V2).
exp1(V,V)                              --> [].

term(V)                                --> factor(V1), term1(V1,V).

term1(V1, tree('*',V1,V2))            --> ['*'], term(V2).
term1(V1, tree('/',V1,V2))            --> ['/'], term(V2).
term1(V,V)                              --> [].

factor(leaf(X))                        --> [X], { number(X) }.
factor(tree(V))                        --> ['('], exp(V), [')'].
```

## Syntax-directed Translation

If the mapping of the source language to the target language is comparatively simple we can employ *syntax-directed translation*.

Syntax-directed translation consist of two phases:

1. build the syntax-tree or structure tree during parsing using an attribute grammar,
2. traverse the syntax-tree recursively generating the target code.

*Example:* Translating arithmetic infix expressions to RPN code

The syntax-directed translation performs a suffix traversal of the parse tree. Note that the translation would have been more complicated if we had directly used the syntax-tree of the left-factored grammar. However, our structure-tree generating DCG has already normalized this representation.

```
:- use_module(library(lists)).  
  
rpn(leaf(X), [X]).  
rpn(tree(Op, V1, V2), Code) :-  
    rpn(V1, Code1),  
    rpn(V2, Code2),  
    append(Code1, Code2, Code3),  
    append(Code3, [Op], Code).
```

Usually syntax-directed translation generates only intermediate code which is then further processed.

## Translation directly in the Grammar

In very simple cases there is no need to split the translation into two separate phases and syntax-directed translation can even be further simplified by unfolding the tree traversal into the attribute grammar.

Keep in mind that even for very simple languages this renders the parser very sensitive to changes in the target language.

*Example:* Translating arithmetic infix expressions to RPN code in a single phase

```
s →      exp
          s.code := append(exp.first, exp.rest);
exp →      term exp'
          exp.first := append(term.first, term.rest);
          exp.rest := append(exp'.first, exp'.rest);
exp' →      +exp
          exp'.first := append(exp.first, [plus]);
          exp'.rest := exp.rest;
exp' →      ε
          exp'.first := [];
          exp'.rest := [];
term →      factor term'
          term.first := factor.code;
          term.rest := append(term'.first, term'.rest);
term' →      *term
          term'.first := append(term.first, [times]);
          term'.rest := term.rest;
term' →      ε
          term'.first := [];
          term'.rest := [];
factor →      int
          factor.code := [push(int.val)]
factor →      (exp)
          factor.code := append(exp.first, exp.code);
```

## Translational Semantics

We can now establish the semantics of some language  $L$  by translating it to some other language  $L'$  of which the semantics is known.

If the mapping from  $L$  to  $L'$  is formalized properly we can regard this as a very simple form of formal semantics.

In our particular case, where we are using logically clean Prolog rules, we are actually establishing this mapping through logical inference rule, so we can regard our mapping from  $L$  to  $L'$  as fully formalized.

## Summary

We have looked at simple syntax directed translation and translational semantics

- Attribute Grammars allow us to perform computations during parsing.
- Particularly, they allow us to obtain an explicit representation of the syntax tree.
- Definite Clause Grammars in PROLOG allow us to implement attribute grammars.
- We can base a syntax-directed translation of a language on a traversal of this syntax tree.
- Establishing a formal mapping from one language to another one with known semantics (via syntax-directed translation) allows us to specify a simple form of translational semantics.

## Homework

- Explore the behaviour of the syntax-directed translator in Prolog given above using tracing.