Programming Language Implementation V

In this lecture we will look at **semantic analysis** and in particular at

- **Attribute grammars**
- **Syntax-directed Translation**
- **Abstract interpretation**

The material is (loosely) based on Aho et al Chapter 5.

Semantic Analysis

Determines those non-syntactic properties that can be determined from the program text. It:
– typically determines the **kind** of each identifier
– performs **type checking** and **type inference**, and
– adds this information to the symbol table.

It also checks that
– variables are declared before use,
– variables are declared only once (within a particular scope),
– type compatibility and required coercions,
– matching of actual with formal parameters,
– resolves overloaded operator symbols
- ...

A separate phase for semantic analysis is required because context-free grammars are not powerful enough to check these properties which they are inherently **context sensitive**. *Remember:* $L = \{wcw \mid w \in (a \mid b)^*\}$, for example, is not a context-free language.

Semantic analysis is often done in a ad hoc manner, but **attribute grammars** can be used to formalize it.

Attribute Grammars

Attribute grammars are due to Knuth in 1968.

They extend the normal grammar mechanism by adding attributes to non-terminals. These attributes can mainly be used for two purposes:

- to compute structures (e.g. syntax-trees) to be returned by the grammar and
- to steer the application of productions by providing additional information when calling a production.

There are two attribute types: *inherited* attributes and *synthesized* attributes. Synthesized attributes are composed by a production. Inherited attributes are provided when a production is called and tested or used to compute synthesized attributes.

Attribute Grammars (cont.)

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

$$b := f(c_1, ..., c_n)$$

where f is a (usually side effect-free) function, c_1, ..., 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 α, 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 $P: X \to X_1, \ldots, X_n$ for a non-terminal X_i are computed from
  1. inherited attributes of X,
  2. synthesized attributes on the right-hand side in $X_1, \ldots, X_{i-1}$,
  3. attributes of terminals on the right-hand side in $X_1, \ldots, X_{i-1}$.
- Synthesized attributes in a production $P: X \to X_1, \ldots, X_n$ 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.

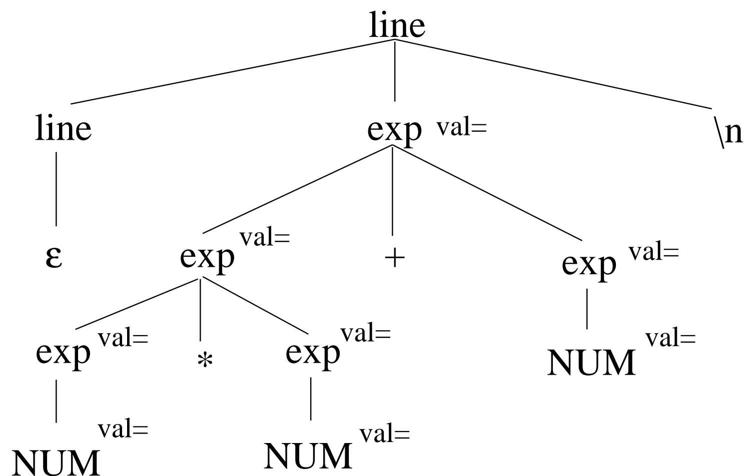
Attribute Grammar – Synthesized

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

Production	Semantic Rules
$line \rightarrow \epsilon$	
$line \rightarrow line\ exp$ `\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 $3 * 5 + 4$`\n` is:



Attribute Grammar – Inherited

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

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

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

Attribute Grammar – Mixed Attributes

In particular grammars that have been left-factored or made LL(k) often loose the "intuitive" structure of the grammars, so that even simple computations require a mix of inherited and syntehsized attributes.

Consider our grammar for arithmetic expressions.

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

Attribute Grammar – Another Example

The following attribution rules determine the type (real or int) of a simple assignment statement:

Grammar rule: $assgn \rightarrow ident := exp$
Attribution Rules:

```
assgn.operation :=
  if ident.type = int then int_asg else real_asg
ident.env := assgn.env
exp.env := assgn.env
```

Condition:

```
coercible(exp.type,ident.type) and
ident.kind = var
```

Grammar rule: $exp1 \rightarrow exp2 + exp3$
Attribution Rules:

```
exp1.type :=
  if exp2.type = int and exp3.type = int
  then int else real
exp1.operation :=
  if exp1.type = int then int_add else real_add
exp2.env := exp1.env
exp3.env := exp1.env
```

Condition:

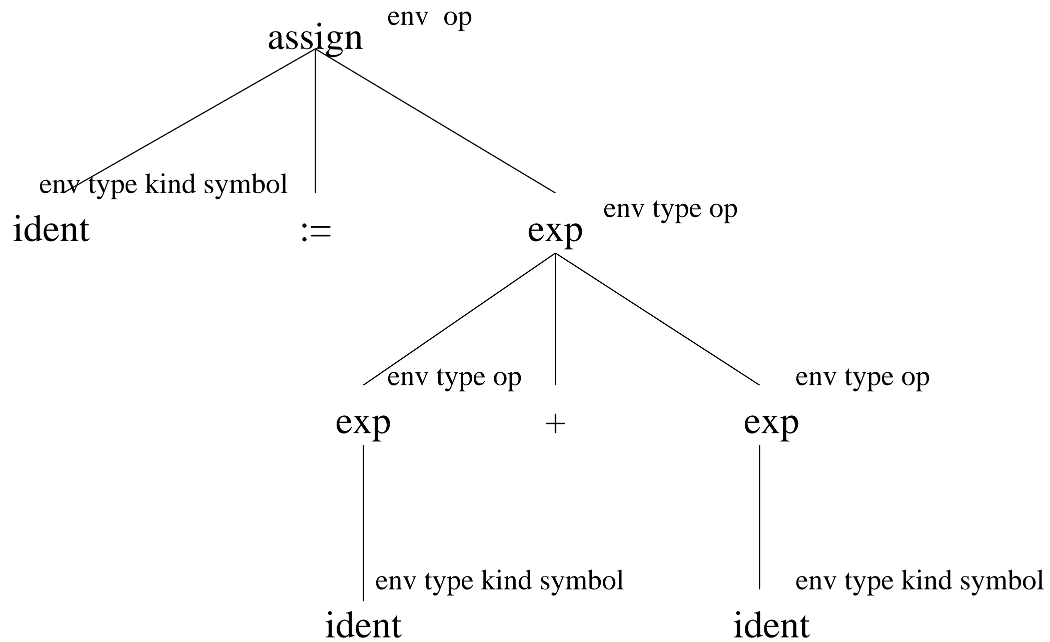
```
coercible(exp2.type,exp1.type) and
coercible(exp3.type,exp1.type)
```

Grammar rule: $exp \rightarrow ident$
Attribution Rules:

```
exp.type :=
  lookup(ident.symbol, exp.env)
```

Attribute Dependency Graph

A sentence such as `x := y+z` has a **dependency graph** detailing how each attribute depends on the other attributes.



Computing Attributes

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**).

However we would like a general (recursive) way of computing attributes while traversing the parse tree.

This **tree traversal** can be specified by giving **visit sequences** for each rule. This details how to traverse the tree and which actions to take at each node during the traversal. The possible steps are:

- visit a child
- visit the parent
- evaluate an attribute
- evaluate a condition.

Computing Attributes – Example

For

Grammar rule: $assgn \rightarrow ident := exp$
Attribution Rules:

```
assgn.operation :=
  if ident.type = int then int_asg else real_asg
ident.env := assgn.env
exp.env := assgn.env
```

Condition:

```
coercible(exp.type,ident.type) and
ident.kind = var
```

we have the visit sequence

```
evaluate ident.env
move to ident
check ident.kind = var
evaluate assgn.operation
evaluate exp.env
move to exp
check coercible(exp.type,ident.type)
```

Question: Do we ever need to visit a node more than once?

Environment Attribute

Conceptually the environment is an inherited attribute. In practice however, it is usually implemented as a single global variable, the **symbol table**. In a language with nested blocks it will be implemented using a stack.

Typical fields are:

- **For a variable:**
  - type and precision
  - address
  - if an array its dimension, subscript and component types
  - if a record its field names and types
  - if a parameter, how it is passed
- **For a type:**
  - its description
- **For a procedure:**
  - whether forward
  - parameter names and types
  - if function, the result type.

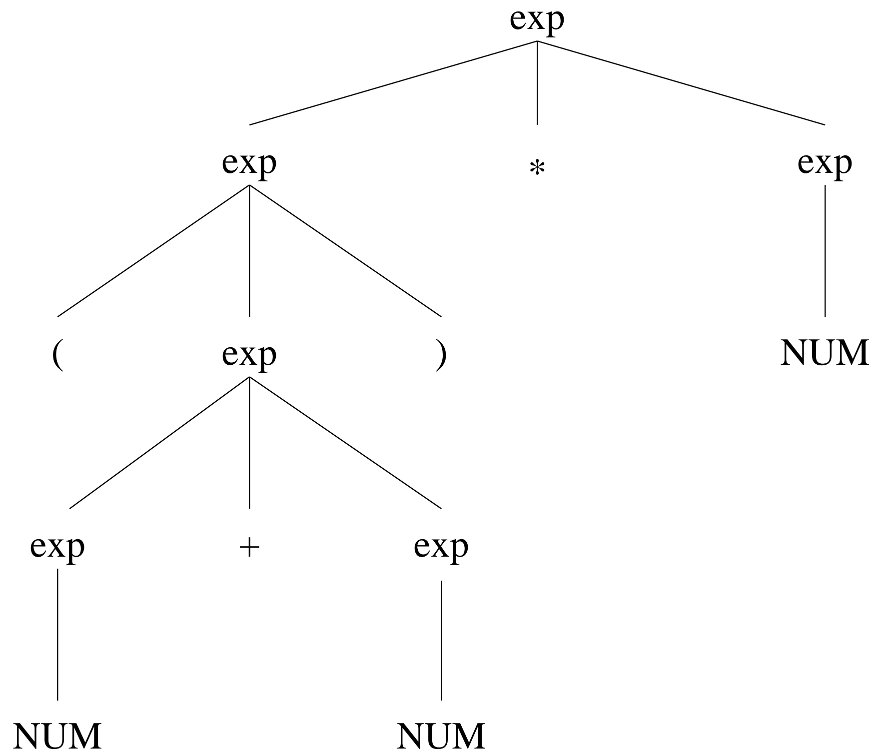
Construction of Structure Trees

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

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:



Attribute Grammar Generators

There are many tools available for automatically generating a semantic analyzer from an attribute grammar. These include

- **Elegant**. The elegant system. Philips Research.
- **Eli**. The eli system, compiler construction made easy. University of Colorado at Boulder, University of Paderborn, Macquarie University in Sydney.
- **FNC-2**. The fnc-2 attribute grammars system. Didier Parigot, Oscar project, INRIA Rocquencourt.
- **FUN**. The fun transformation system. Attribute Grammar Based Transformation Systems.

Search the Web if you are interested. A good starting point is:
`http://www-rocq.inria.fr/oscar/www/fnc2/attribute-grammar-people.html`

Abstract Interpretation

Abstract interpretation is another approach to semantic analysis.

It formalises **approximate** computation like the "rule of signs."

The idea of performing program analysis by approximate computation appeared very early in computer science.
– Naur identified the idea and used it in the Gier Algol compiler in the early sixties.
– It was formalized by Cousot & Cousot in 1977.

The key idea is to run the program with **descriptions** of data rather than the data itself.

For instance, the *Sign* descriptions are

$\ominus$ — the negative integers
$\oplus$ — the positive integers
0 — 0
$\top$ — any integer

You must replace operations performed on data by **abstract operations**.

For instance, abstract multiplication with *Sign*s is

$mult_{Sign}$	$\ominus$	0	$\oplus$	$\top$
$\ominus$	$\oplus$	0	$\ominus$	$\top$
0	0	0	0	0
$\oplus$	$\ominus$	0	$\oplus$	$\top$
$\top$	$\top$	0	$\top$	$\top$

These need to be **safe**, i.e. correct, approximations.

Abstract Interpretation (Cont.)

What is the definition of abstract addition?

$plus_{Sign}$	$\ominus$	0	$\oplus$	$\top$
$\ominus$				
0				
$\oplus$				
$\top$				

Analyze `fac` using $Sign$:

```
int fac(int n)
{ int fac;

  fac = 1;
  while (n > 1) {
    fac = fac * n;
    n = n + (-1);
  }
  return fac;
}
```

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

First we need to declare a datatype for the instruction list.

```
datatype instruction =
   PUSH of result | TIMES_OP | DIV_OP | PLUS_OP | SUB_OP ;
```

The syntax-directed translation performs a suffix traversal of the parse tree. Note how the left-factored structure of the grammar forces us to process the code in a rather unnatural fashion: every production generates two fragments of code - the `first` argument which essentially represents the fragment completing the last arithmetic expression/term and the `rest` argument representing the remainder of the expression or term.

Syntax-directed Translation (cont.)

```
fun genCode Empty = ([], [])
 |  genCode (EXP(termTree, exp1Tree)) =
      let val (tFirst, tRest) = genCode termTree in
       let val (eFirst, eRest) = genCode exp1Tree in
        ((tFirst @ tRest), (eFirst @ eRest))
       end
      end
 |  genCode (EXP1(OP(PLUS), expTree)) =
      let val (first, Rest) = genCode expTree in
        ((first @ [PLUS_OP]), Rest)
      end
 |  genCode (EXP1(OP(SUB), expTree)) =
      let val (first, Rest) = genCode expTree in
        ((first @ [SUB_OP]), Rest)
      end
 |  genCode (TERM(factorTree, term1Tree)) =
      (* note that the factor could in fact be
         an expression in brackets. In this case
         we need to collect first, too
      *)
      let val (fFirst,fRest) = genCode factorTree in
       let val (tFirst, tRest) = genCode term1Tree in
         ((fFirst @ fRest), (tFirst @ tRest))
       end
      end
```

...continued on next page

Syntax-directed Translation (cont.)

...continued from previous page

```
 |  genCode (TERM1(OP(DIV), termTree)) =
      let val (first, rest) = genCode termTree in
        ((first @  [DIV_OP]), rest)
      end
 |  genCode (TERM1(OP(TIMES), termTree)) =
      let val (first, rest) = genCode termTree in
        ((first @ [TIMES_OP]), rest)
      end
 |  genCode (FACTOR(X)) = ([], [PUSH(X)]);

fun translate () =
  let val (expFirst, expRest) = genCode(parse()) in
    (expFirst @ expRest)
  end
```

For more complex languages, 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 int the target language.

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

$$
\begin{array}{ll}
s \rightarrow & exp \\
 & s.code := append(exp.first, exp.rest); \\
exp \rightarrow & term\ exp' \\
 & exp.first := append(term.first, term.rest); \\
 & exp.rest := append(exp'.first, exp'.rest); \\
exp' \rightarrow & +exp \\
 & exp'.first := append(exp.first, [plus]); \\
 & exp'.rest := exp.rest; \\
exp' \rightarrow & \epsilon \\
 & exp'.first := []; \\
 & exp'.rest := []; \\
term \rightarrow & factor\ term' \\
 & term.first := factor.code; \\
 & term.rest := append(term'.first, term'.rest); \\
term' \rightarrow & *term \\
 & term'.first := append(term.first, [times]); \\
 & term'.rest := term.rest; \\
term' \rightarrow & \epsilon \\
 & term'.first := []; \\
 & term'.rest := []; \\
factor \rightarrow & \mathbf{int} \\
 & factor.code := [push(int.val)] \\
factor \rightarrow & (exp) \\
 & factor.code := append(exp.first, exp.code);
\end{array}
$$

Summary

We have looked at semantic analysis, attribute grammars, abstract interpretation and syntax-directed translation.

Homework

- Read Chapter 5 of Aho et al.
- Extend the assignment grammar example and give attribute rules and conditions for the productions

  $exp \rightarrow exp - exp$,

  $exp \rightarrow exp\ div\ exp$,

  $exp \rightarrow floor(exp)$,

  where div is integer division and $floor$ takes a real and returns an integer.
- Give a description domain for analyzing the *parity* of an integer, i.e. if it is odd or even.
  Give abstract operations for multiplication and addition.