4. Semantic Processing and Attributed Grammars
Semantic Processing

The parser checks only the *syntactic* correctness of a program.

**Tasks of semantic processing**

- **Checking context conditions**
  - Declaration rules
  - Type checking

- **Symbol table handling**
  - Maintaining information about declared names
  - Maintaining information about types
  - Maintaining scopes

- **Invocation of code generation routines**

Semantic actions are integrated into the parser. We describe them with *attributed grammars*. 
**Semantic Actions**

So far, we have just *analyzed* the input

```plaintext
Number = digit {digit}.
```

the parser checks if the input is syntactically correct
(in this example *Number* is not viewed as part of the
lexical structure of the language)

Now, we also *translate* it (semantic processing)

e.g.: we want to count the digits in the number

```plaintext
Number =

digit (. int n = 1; .)
{
    digit (. n++; .)
}

(. System.out.println(n); .)
```

*semantic actions*

• arbitrary Java statements between (. and .)
• are executed by the parser at the position
  where they occur in the grammar

"translation" here:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>3</td>
</tr>
<tr>
<td>4711</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
Attributes

Syntax symbols can return values (sort of output parameters)

```
digit <\uparrow val>  \quad digit \text{ returns its numeric value (0..9) as an output attribute}
```

Attributes are useful in the translation process

e.g.: we want to compute the value of a number

```
Number = digit <\uparrow val>
{  digit <\uparrow n>
}
{\text{. System.out.println(val); .}}
```

"translation" here:

"123"  \implies  123
"4711"  \implies  4711
"9"  \implies  9
Input Attributes

Nonterminal symbols can have also input attributes (parameters that are passed from the "calling" production)

\[ \text{Number} \downarrow \text{base}, \uparrow \text{val} \]

- \textit{base}: number base (e.g. 10 or 16)
- \textit{val}: returned value of the number

Example

\[
\text{Number} \downarrow \text{base}, \uparrow \text{val} \quad \text{( \text{int base, val, n; } .)}
\]

= \text{digit} \uparrow \text{val}

\{
\text{digit} \uparrow \text{n}
\}

\text{( \text{val = base * val + n; } .)}
Attributed Grammars

Notation for describing translation processes consist of three parts

1. Productions in EBNF
   \[ \text{IdentList} = \text{ident} \{"","\text{ident}\}. \]

2. Attributes (parameters of syntax symbols)
   \[
   \begin{align*}
   \text{ident} & \downarrow \text{name} \\
   \text{IdentList} & \downarrow \text{type}
   \end{align*}
   \]
   output attributes (synthesized): yield the translation result
   input attributes (inherited): provide context from the caller

3. Semantics actions
   
   \[
   (\ldots \text{arbitrary Java statements} \ldots )
   \]
Example

ATG for processing declarations

VarDecl = Type <↑type> IdentList <↓type> ";" .

IdentList <↓type> = ident <↑name> 
{ "," ident <↑name> 
} .

This is translated to parsing methods as follows

private static void VarDecl() {
    Struct type;
    type = Type();
    IdentList(type);
    check(semicolon);
}

private static void IdentList(Struct type) {
    String name;
    check(ident); name = t.string;
    Tab.insert(name, type);
    while (sym == comma) {
        scan();
        check(ident); name = t.string;
        Tab.insert(name, type);
    }
}

ATGs are shorter and more readable than parsing methods.
Example: Processing of Constant Expressions

input: $3 \times (2 + 4)$
desired result: 18

```
Expr \(<\uparrow val>\) = Term \(<\uparrow val>\)
   { "+" Term \(<\uparrow val>\) ( . int val, val1 ; . )
     | "-" Term \(<\uparrow val>\) ( . val = val - val1 ; . )
   }. 

Term \(<\uparrow val>\) = Factor \(<\uparrow val>\)
   { "*" Factor \(<\uparrow val>\) ( . val = val * val1 ; . )
     | "/" Factor \(<\uparrow val>\) ( . val = val / val1 ; . )
   }.

Factor \(<\uparrow val>\) = number ( . int val, val1 ; . )
   | "(" Expr \(<\uparrow val>\) ")" ( . int val, val1 ; . )
```

Diagram:
```
Expr  ↑18
   ↑18
Term  ↑18
  ↑6
Factor  ↑6
   ↑2
  ↑2
Factor  ↑2
   ↑4
  ↑4
Factor  ↑4
  ↑3
   ↑3
3  *  ( 2 + 4 )  4
```
Transforming an ATG into a Parser

Production

\[
\text{Expr} \leftarrow \text{Term} \leftarrow \text{Term} \leftarrow (\text{int val}, \text{val1}; .)
\]

Parsing method

```java
private static int Expr() {
    int val, val1;
    val = Term();
    for (;;) {
        if (sym == plus) {
            scan();
            val1 = Term();
            val = val + val1;
        } else if (sym == minus) {
            scan();
            val1 = Term();
            val = val - val1;
        } else break;
    }
    return val;
}
```

- **input attributes** ⇒ **parameters**
- **output attribute** ⇒ **function value**
  (if there are multiple output attributes encapsulate them in an object)
- **semantic actions** ⇒ **embedded Java code**

Terminal symbols have no input attributes.
In our form of ATGs they also have no output attributes,
but their value can be obtained from \text{t.string} or \text{t.val}. 
Example: Sales Statistics

ATGs can also be used in areas other than compiler construction

Example: given a file with sales numbers

File = {Article}.
Article = Code {Amount} ";"
Code = number.
Amount = number.

Whenever the input is syntactically structured
ATGs are a good notation to describe its processing

Input for example:

3451 2 5 3 7 ;
3452 4 8 1 ;
3453 1 1 ;
...

Desired output:

3451 17
3452 13
3453 2
...

ATG for the Sales Statistics

File = { Article <\text{code}, \text{amount}> }.  

Article <\text{code}, \text{amount}> 
= Value <\text{code}> 
{ Value <\text{x}> 
} "." .  

Value <\text{x}> 
= number .  

Parser code

```java
private static void File() {
    while (sym == number) {
        ArtInfo a = Article();
        print(a.code + " " + a.amount);
    }
}

class ArtInfo {
    int code, amount;
}

private static ArtInfo Article() {
    ArtInfo a = new ArtInfo();
    a.code = Value(); a.amount = 0;
    while (sym == number) {
        int x = Value();
        a.amount += x;
    }
    check(semicolon); return a;
}

private static int Value() {
    check(number);
    return t.val;
}

terminal symbols
number
semicolon
eof
```

Code and Amount can be merged
The attributes capture the difference in semantics.
**Example: Image Description Language**

**described by:**

```
POLY
(10,40)
(50,90)
(40,45)
(50,0)
END
```

**input syntax:**

```
Polygon = "POLY" Point {Point} "END".
Point = "(" number "," number ")".
```

We want a program that reads the input and draws the polygon

```
Polygon = "POLY"
Point(p, q; .)
"END"
Point(p; int x, y; .)
```

We use "Turtle Graphics" for drawing

- **Turtle.start(p);** sets the turtle (pen) to point \( p \)
- **Turtle.move(q);** moves the turtle to \( q \)
  drawing a line
**Example: Transform Infix to Postfix Expressions**

Arithmetic expressions in infix notation are to be transformed to postfix notation:

- $3 + 4 \times 2 \Rightarrow 3 4 2 \times +$
- $(3 + 4) \times 2 \Rightarrow 3 4 + 2 \times$

**Expr**

```
Expr = Term
{ "+" Term (. print("+"); .) | "-" Term (. print("-"); .) }
```

**Term**

```
Term = Factor
{ "*" Factor (. print("*"); .) | "/" Factor (. print("/"); .) }
```

**Factor**

```
Factor = number (. print(t.val); .) | "(" Expr ")"
```