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

So far, we have just **analyzed** the input

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number = digit { digit }</td>
<td>(. int n = 1; .) (. n++; .) (. System.out.println(n); .)</td>
</tr>
</tbody>
</table>

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

"translation" here:

123  ⇒  3
4711 ⇒  4
9    ⇒  1
Attributes

Syntax symbols can return values (sort of output parameters)

digit $<\hat{\text{val}}>\quad$ *digit* 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 $<\hat{\text{val}}>$
  { digit $<\hat{n}>$
    { digit $<\hat{n}>$
      (. val = 10 * val + n; .)
    }
  }
  (. System.out.println(val); .)
```

"translation" here:

"123" $\Rightarrow$ 123
"4711" $\Rightarrow$ 4711
"9" $\Rightarrow$ 9
Input Attributes

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

Number $<\downarrow \text{base, } \uparrow \text{val}>$

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

Example

Number $<\downarrow \text{base, } \uparrow \text{val}>$ = digit $<\uparrow \text{val}>$
{ digit $<\uparrow \text{n}>$
  (. 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)

   
   \[
   \text{ident}<\text{name}> \quad \text{output attributes (synthesized): yield the translation result}
   \]
   \[
   \text{IdentList}<\downarrow \text{type}> \quad \text{input attributes (inherited): provide context from the caller}
   \]

3. Semantic actions

   
   \[
   (\ldots \text{arbitrary Java statements} \ldots )
   \]
Example

ATG for processing declarations

```
VarDecl ( . Struct type; . )
  = Type <↑type>
    IdentList <↓type>
    ""," .

IdentList <↓type> ( . Struct type; String name; . )
  = 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

\[
\text{Expr} \leftarrow^\text{val} = \text{Term} \leftarrow^\text{val} \\
\{ "+" \text{ Term} \leftarrow^\text{val1} | "-" \text{ Term} \leftarrow^\text{val1} \}
\]

\[
\text{Term} \leftarrow^\text{val} = \text{Factor} \leftarrow^\text{val} \\
\{ "\times" \text{ Factor} \leftarrow^\text{val1} | "/" \text{ Factor} \leftarrow^\text{val1} \}
\]

\[
\text{Factor} \leftarrow^\text{val} = \text{number} | \("\) \text{ Expr} \leftarrow^\text{val} \"
\]
Transforming an ATG into a Parser

Production

```plaintext
Expr <\uparrow val> = Term <\uparrow val> 
{ "+" Term <\uparrow val1> ( . val = val + val1; . ) 
| "-" Term <\uparrow val1> ( . val = val - 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 `t.string` or `t.val`. 

SSW


**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 <code, amount> 
    }.

Article <code, amount>
= Number <code>
    { Number <x>
        (. amount += x; .)
        ",".
    }

Number <x>
= number
    (. x = t.val; .)

Parser code

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.amount = 0;
    a.code = Number();
    while (sym == number) {
        int x = Number();
        a.amount += x;
    }
    check(semicolon); return a;
}

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

terminal symbols
number
semicolon
eof
Example: Image Description Language

described by:

input syntax:

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

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

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

Poly
蓬 "POLY"
蓬 Point<蓬>p>蓬 
蓬{蓬"," Point<蓬>q>
蓬 }
蓬"END"
蓬 .
蓬 Point<蓬>p>
蓬 = "(" number 
蓬 "," number 
"
蓬 ")"
蓬 .

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:

\[ 3 + 4 \times 2 \Rightarrow 3 4 2 \times + \]
\[ (3 + 4) \times 2 \Rightarrow 3 4 + 2 \times \]

\[
\text{Expr} = \text{Term} \{
\text{"+" Term} \quad \text{(. print("+"); .)}
| \text{"-" Term} \quad \text{(. print("-"); .)}
\}.
\]

\[
\text{Term} = \text{Factor} \{
\text{"*" Factor} \quad \text{(. print("*"); .)}
| \text{"/" Factor} \quad \text{(. print("/"); .)}
\}.
\]

\[
\text{Factor} = \text{number} \quad \text{(. print(t.val); .)}
| \text{"(" Expr ")"}.
\]