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

```
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

```
Number =
    digit     (. int n = 1; .)
    { digit   (. n++; .)
    }
              (. System.out.println(n); .)
    .
```

syntax | semantics

semantic actions

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

"translation" here:

123 Þ 3
4711 Þ 4
9 Þ 1

Attributes

Syntax symbols can return values (sort of output parameters)

`digit <- val>` *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              (. int val, n; .)
=  digit <- val>
   { digit <- n>    (. val = 10 * val + n; .)
   }
                    (. System.out.println(val); .)
   .
```

"translation" here:

"123" Þ 123
"4711" Þ 4711
"9" Þ 9

Input Attributes

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

```
Number <¯base, - val>
```

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

Example

```
Number <¯base, - val>    (. int base, val, n; .)
= digit <- val>
  { digit <- n>          (. val = base * val + n; .)
  }.
```

Attributed Grammars

Notation for describing translation processes

consist of three parts

1. Productions in EBNF

```
IdentList = ident {"," ident}.
```

2. Attributes (parameters of syntax symbols)

ident<- name>	output attributes (*synthesized*):	yield the translation result
IdentList<¯type>	input attributes (*inherited*):	provide context from the caller

3. Semantic actions

```
(.   ... arbitrary Java statements ...   .)
```

Example

ATG for processing declarations

```
VarDecl                      (. Struct type; .)
=  Type <- type>
   IdentList <¯type>
   ";" .
```

```
IdentList <¯type>            (. Struct type; String name; .)
=  ident <- name>            (. Tab.insert(name, type); .)
   { "," ident  <- name>     (. Tab.insert(name, type); .)
   } .
```

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

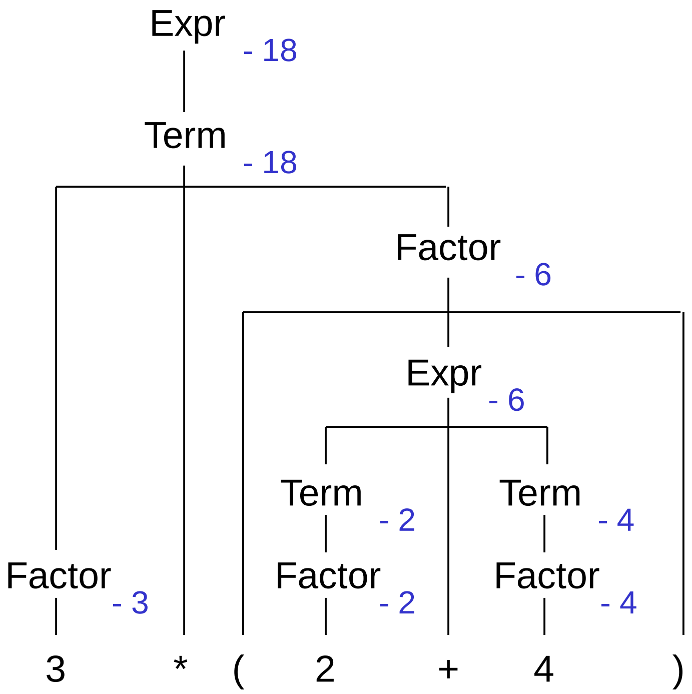
Example: Processing of Constant Expressions

input: 3 * (2 + 4)
desired result: 18

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

Term <- val>                  (. int val, val1; .)
= Factor <- val>
  { "*" Factor <- val1>       (. val = val * val1; .)
  | "/" Factor <- val1>       (. val = val / val1; .)
  }

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



Transforming an ATG into a Parser

Production

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

Parsing method

```
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	Þ	parameters
output atribute	Þ	function value (if there are multiple output attributes encapsulate them in an object)
semantic actions	Þ	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*.

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 syntacticlly 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                                (. int code, amount; .)
= { Article <- code, - amount>      (. print(code + " " + amount); .)
  }.

Article <- code, - amount>          (. int code, x, amount = 0; .)
=  Number <- code>
   { Number <- x>                   (. amount += x; .)
   }
   ";".

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

Parser code

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

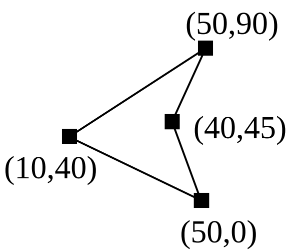
```
class ArtInfo {
   int code, amount;
}
```

```
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:

```
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                 (. Pt p, q; .)
= "POLY"
   Point<- p>           (. Turtle.start(p); .)
   { "," Point<- q>     (. Turtle.move(q); .)
   }
   "END"                (. Turtle.move(p); .)
   .

Point<- p>              (. Pt p; int x, y; .)
= "(" number            (. x = t.val; .)
   "," number           (. y = t.val; .)
   ")"                  (. p = new Pt(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 * 2 Þ 3 4 2 * +

(3 + 4) * 2 Þ 3 4 + 2 *

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

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

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