4. Semantic Processing and Attribute Grammars

Semantic Processing

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

Tasks of semantic processing

- **Symbol table handling**
  - Maintaining information about declared names
  - Maintaining information about types
  - Maintaining scopes
- **Checking context conditions**
  - Scoping rules
  - Type checking
- **Invocation of code generation routines**

Semantic actions are integrated into the parser
and are described with *attribute grammars*

Semantic Actions

So far: *analysis* of the input

```
Expr = Term { "+" Term }.
```

the parser checks if the input is syntactically correct.

Now: *translation* of the input (semantic processing)

e.g.: we want to count the terms in the expression

```
Expr =
   Term          (. int n = 1; .)
   { "+" Term  (. n++; .)
   }               (. Console.WriteLine(n); .)
   .
```

semantic actions

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

"translation" here:

1+2+3 ⇒ 3
47+1 ⇒ 2
909 ⇒ 1

Attributes

Syntax symbols can return values (sort of output parameters)

`Term <↑int val>` *Term* returns its numeric value as an output attribute

Attributes are useful in the translation process

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

```
Expr                    (. int sum, val; .)
= Term<↑sum>
   { "+" Term<↑val>     (. sum += val; .)
   }                    (. Console.WriteLine(sum); .)
   .
```

"translation" here:

1+2+3 ⇒ 6

47+1 ⇒ 48

909 ⇒ 909

Input Attributes

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

```
Expr<↓bool printHex>
```

printHex: print the result of the addition hexadecimal
(otherwise decimal)

Example

```
Expr<↓bool printHex>        (. int sum, val; .)
= Term<↑sum>
  { "+" Term<↑val>          (. sum += val; .)
  }.                        (.      if (printHex) Console.WriteLine("{0:X}", sum)
                                    else Console.WriteLine("{0:D}", sum);
                             .)
```

Attribute Grammars

Notation for describing translation processes

consist of three parts

1. Productions in EBNF

```
Expr = Term { "+" Term }.
```

2. Attributes (parameters of syntax symbols)

`Term<↑int val>`	output attributes (*synthesized*):	yield the translation result
`Expr<↓bool printHex>`	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 <↓Struct type>
=  ident                  (. Tab.insert(token.str, type); .)
   { "," ident            (. Tab.insert(token.str, type); .)
   } .
```

This is translated to parsing methods as follows

```
static void VarDecl () {
   Struct type;
   Type(out type);
   IdentList(type);
   Check(Token.SEMICOLON);
}
```

```
static void IdentList (Struct type) {
   Check(Token.IDENT);
   Tab.Insert(token.str, type);
   while (la == Token.COMMA) {
      Scan();
      Check(Token.IDENT);
      Tab.Insert(token.str, type);
   }
}
```

ATGs are shorter and more readable than parsing methods

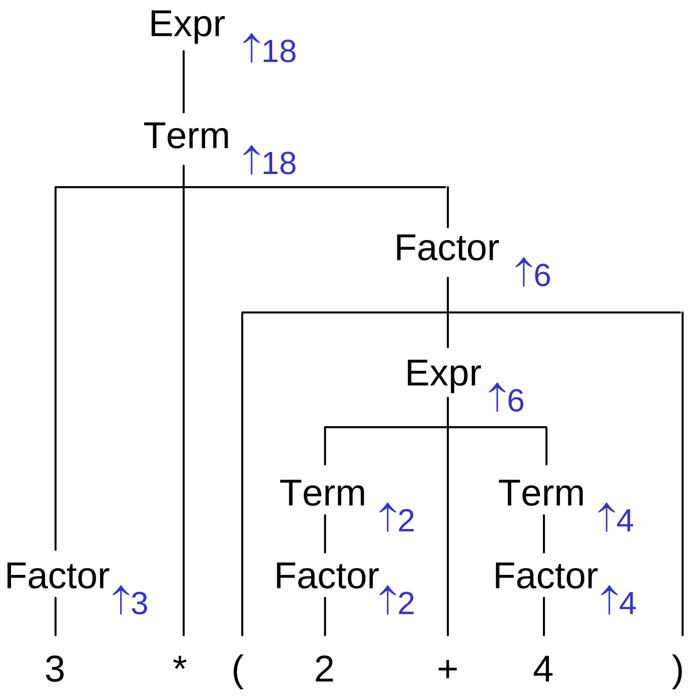
Example: Processing of Constant Expressions

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

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

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

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



Transforming an ATG into a Parser

Production

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

Parsing method

```
static void Expr (out int val) {
   int val1;
   Term(out val);
   for (;;) {
      if (la == Token.PLUS) {
         Scan();
         val1 = Term(out val1);
         val += val1;
      } else if (la == Token.MINUS) {
         Scan();
         Term(out val1);
         val -= val1;
      } else break;
   }
}
```

input attribute ⇒ parameter
output atribute ⇒ *out* parameter
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 is computed from *token.str or token.val*.

Example: Sales Statistics

ATGs can also be used in areas other than compiler constructions

Example: given a file with sales numbers

```
File    = { Article }.
Article = Code { Amount } "END"
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  END
3452    4  8  1 END
3453    1  1 END
...
```

Desired output:

```
3451    17
3452    13
3453    2
...
```

ATG for the Sales Statistics

```
File                              (. int code, amount; .)
= { Article<↑code, ↑amount>       (. Write(code + " " + amount); .)
  }.

Article<↑int code, ↑int amount>
=  Value<↑code>
   {                              (. int x; .)
      Value<↑x>                   (. amount += x; .)
   }
   "END".

Value<↑int x>
= number                          (. x = token.val; .)
   .
```

Parsercode

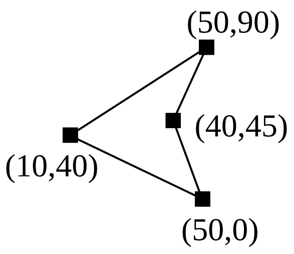
```
static void File () {
   int code, amount;
   while (la == number) {
      Article(out code, out number);
      Write(code + " " + amount);
   }
}
```

```
static void Article (out int code, out int amount) {
   Value(out code);
   while (la == number) {
      int x; Value(out x); amount += x;
   }
   Check(end);
}
```

```
static void Value (out int x) { Check(number); x = token.val; }
```

terminal symbols
number, end, 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 ⇒ 3 4 2 * +
(3 + 4) * 2 ⇒ 3 4 + 2 *

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

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

Factor =
   number        (. Write(token.val); .)
|  "(" Expr ")".
```