4. Syntax-Directed Translation

Emil Sekerinski, McMaster University, February 2022

> This notebook contains type hints that allow type-checking with mypy. See also this introduction, the Python typing library, and this cheat sheet. The nb_mypy notebook extension type-checks notebook cells with mypy as they are executed. The extension can be installed by `python3 -m pip install nb_mypy`, which also installs mypy, and then has to be enabled by running the line magic below.

In [ ]:
```
%load_ext nb_mypy
```

Attribute Grammars

So far we were only concerned with accepting or rejecting the input according to a grammar. The goal is, of course, to produce eventually output, in the case of a compiler to generate machine code. To this end, we use *attribute grammars*. These attach computation to a parse tree. Attribute grammars extend context-free grammars by

- associating a set of named *attributes* with each symbol and
- augmenting productions with *attribute evaluation rules*.

To every symbol `X` of a grammar, a computation is associated that computes the attributes of `X`. Productions are of the form

$$X(s_1,s_2) \to \ldots\ Y(t_1,\ t_2)\ \ldots\ Z(u_1,\ u_2)\ \ldots$$

where $s_1,\ s_2,\ t_1,\ t_2,\ u_1,\ u_2$ are the attributes associated with the corresponding symbols. The computation, in its simplest form, is a function that computes the attributes on the left-hand side of a production from the attributes on the right-hand side:

$$(s_1,\ s_2) = f(t_1,\ t_2,\ u_1,\ u_2)$$

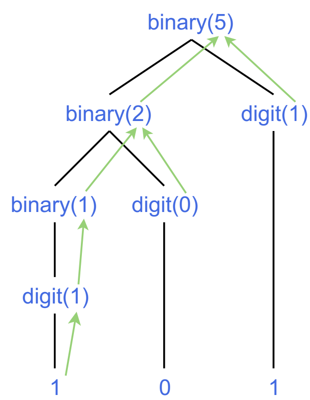
With an implementation in mind, we allow not only mathematical functions but programming language statements to express the computation. If a symbol appears multiple times in a production, the attributes are given unique names.

Consider a grammar for binary numbers with productions:

```
binary → binary digit
binary → digit
digit → '0'
digit → '1'
```

For computing the value of a binary number, one integer attribute is associated with `digit` and one integer attribute with `binary`. An attribute grammar computing the value is:

production	attribute rule
$\texttt{binary}(v_0) \to \texttt{binary}(v_1)\ \texttt{digit}(v_2)$	$v_0 := 2 \times v_1 + v_2$
$\texttt{binary}(v_0) \to \texttt{digit}(v_1)$	$v_0 := v_1$
`digit(v) → '0'`	`v := 0`
`digit(v) → '1'`	`v := 1`



In the parse tree, the attributes are evaluated bottom-up, indicated by the green arrows in the figure to the right. The meaning of a sentence is given by the attributes of the start symbol from which it is derived.

Above grammar has a left-recursive production, making it unsuitable for recursive descent parsing. An equivalent grammar in EBNF is:

```
binary → digit { digit }
digit → '0' | '1'
```

With EBNF, the attribute rules are placed "inside" the productions to express that a rule is to be executed after the preceding nonterminal is recognized, as would be with the plain grammar. The attribute rules are delineated by `«` and `»`:

```
binary(v₀) → digit(v₁) « v₀ := v₁ » { digit(v₂) « v₀ := 2 × v₀ + v₂ » }
digit(v) → '0' « v := 0 » | '1' « v := 1 »
```

As the assignment $v_0 := v_1$ only copies a value, it can be omitted by renaming the attributes. Now the first `digit` in the productions for `binary` assigns the initial value of `v`:

```
binary(v) → digit(v) { digit(w) « v := 2 × v + w » }
digit(v) → '0' « v := 0 » | '1' « v := 1 »
```

Let us define the EBNF of EBNF with attributes:

```
grammar  →  production {'\n' production }
production  →  identifier attributes '→' expression
expression  →  term { '|' term }
term  →  factor { ' ' factor }
factor  →  (identifier attributes | string | '(' expression ')' | '[' expression ']' | '{'
expression '}') [rule]
rule  →  '«' statement '»'
attributes  →  [ '(' identifier {',' identifier} ')' ]
identifier  →  letter { letter | digit }
letter  →  'A' | … | 'Z'
digit  →  '0' | … | '9'
string  →  '\'' { char } '\''
```

In the construction of a recursive descent parser, the attributes become result parameters of the parsing procedures. (The notation $\texttt{procedure p}(v_1, v_2) \rightarrow (r_1, r_2)$ is used for a procedure with value parameters v_1, v_2 and result parameters r_1, r_2.) The rule for constructing parsing procedures is extended by having productions with list of attributes, `as` below:

p	pr(p)
`B(as) → E`	`procedure B() → (as)` `pr(E)`

The rules for constructing `pr(E)` are extended to include attribute evaluation rules:

E	pr(E)
`«stat»`	`stat`

As Python does not have named result parameters, local variables for the attributes are introduced and returned at the end of each parsing procedure. Here is the parser for above grammar, first without attribute rules:

```python
src: str; pos: int; sym: str

def nxt():
    global pos, sym
    if pos < len(src): sym, pos = src[pos], pos + 1
    else: sym = chr(0) # end of input symbol

def binary(): # binary → digit { digit }
    digit()
    while sym in '01': digit()

def digit(): # digit → '0' | '1'
    if sym == '0': nxt()
    elif sym == '1': nxt()
    else: raise Exception("invalid character at " + str(pos))

def parse(s: str):
    global src, pos;
    src, pos = s, 0; nxt(); binary()
    if sym != chr(0): raise Exception("unexpected character at " + str(pos))
```

A trace of the calling sequence is given in case of incorrect input, otherwise there is no effect:

```python
#parse("2") # invalid character at 1
#parse("0x1") # unexpected character at 2
parse("101")
```

Here is the parser with attribute rules added:

```python
src: str; pos: int; sym: str

def nxt():
    global pos, sym
    if pos < len(src): sym, pos = src[pos], pos + 1
```

```
    else: sym = chr(0) # end of input symbol

def binary() -> int: # binary(v) → digit(v) { digit(w) « v := 2 × v + w » }
    v = digit()
    while sym in '01': w = digit(); v = v * 2 + w
    return v

def digit() -> int: # digit(v) → '0' « v := 0 » | '1' « v := 1 »
    if sym == '0': nxt(); w = 0
    elif sym == '1': nxt(); w = 1
    else: raise Exception("invalid character at " + str(pos))
    return w

def evaluate(s: str) -> int:
    global src, pos;
    src, pos = s, 0; nxt(); v = binary()
    if sym != chr(0): raise Exception("unexpected character at " + str(pos))
    return v
```

In [ ]:
```
evaluate("101") # returns 5
```

Evaluating Arithmetic Expressions

Consider *evaluating arithmetic expressions* over constant integers. In the following expression grammar, the symbols are characters and white space (`ws`) is allowed around operators and integers:

```
expression → ws term { '+' ws term }
term → factor { '*' ws factor }
factor → integer | '(' expression ')' ws
integer → digit { digit } ws
digit → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
ws → { ' ' }
```

Attribute rules are added for evaluating an expression:

```
expression(v) → ws term(v) { '+' ws term(w) « v := v + w » }
term(v) → factor(v) { '*' ws factor(w) « v := v × w » }
factor(v) → integer(v) | '(' expression(v) ')' ws
integer(v) → digit(v) { digit(w) « v := 10 × v + w » } ws
digit(v) → '0' « v := 0 » | … | '9' « v := 9 »
ws → { ' ' }
```

The implementation below contains several simplifications:

- the test `sym ∈ {'0', '1', … '9'}` is implemented by `'0' <= sym <= '9'`,
- if `sym` is a digit, it is converted to an integer by `ord(sym) - ord('0')`.

In [ ]:
```
src: str; pos: int; sym: str

def nxt():
    global pos, sym
    if pos < len(src): sym, pos = src[pos], pos + 1
    else: sym = chr(0) # end of input symbol

def expression() -> int: # expression → ws term { '+' ws term }
    ws(); v = term()
    while sym == '+': nxt(); ws(); w = term(); v = v + w
    return v

def term() -> int: # term → factor { '*' ws factor }
    v = factor()
    while sym == '*': nxt(); ws(); w = factor(); v = v * w
    return v

def factor() -> int: # factor → integer | '(' expression ')' ws
    if '0' <= sym <= '9': v = integer()
    elif sym == '(':
        nxt(); v = expression()
        if sym == ')': nxt(); ws()
        else: raise Exception("')' expected at " + str(pos))
    else: raise Exception("invalid character at " + str(pos))
    return v

def integer() -> int: # integer(v) → digit(v) { digit(w) « v := 10 * v + w » } ws
    # '0' <= sym <= '9'
    v = digit()
    while '0' <= sym <= '9': v = 10 * v + digit()
    ws()
```

```
    return v

def digit() -> int: # integer → digit { digit } ws
    # '0' <= sym <= '9'
    v = ord(sym) - ord('0'); nxt()
    return v

def ws(): # ws → { ' ' }
    while sym == ' ': nxt()

def evaluate(s: str) -> int:
    global src, pos;
    src, pos = s, 0; nxt(); v = expression()
    if sym != chr(0): raise Exception("unexpected character at " + str(pos))
    return v
```

In [ ]:
```
#evaluate("(2 + 3") # ')' expected at 6
#evaluate("2 + x") # invalid character at 5
#evaluate("2 + 3!") # unexpected character at 6
evaluate("(2 + 3) * 4 + 5")
```

Type-checking

Attribute grammars can be used for *type-checking*. To illustrate this, relational operators `=`, `<` and boolean operators `&` (conjunction), `|` (disjunction) are introduced:

```
expression → ws simpleExpression [ ( '=' | '<' ) ws simpleExpression]
simpleExpression → ws term { ('+' | '|' ) ws term }
term → factor { ( '*' | '&' ) ws factor }
factor → integer | '(' expression ')' ws
integer → digit { digit } ws
digit → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
ws → { ' ' }
```

Values are of type `int` or `bool`. Operators are of following types, where `t` is either `int` or `bool`:

```
= : t × t → bool
< : int × int → bool
+ : int × int → int
* : int × int → int
| : bool × bool → bool
& : bool × bool → bool
```

The attribute grammar below checks the types and either evaluates the expression if it is type-correct or flags an error:

```
expression(v) → ws simpleExpr(v)
    [ '=' ws simpleExpr(w) «if type(v) = type(w) then v := v = w else error»
    | '<' ws simpleExpr(w) «if type(v) = int = type(w) then v := v < w else error» ]
simpleExpr(v) → term(v)
    { '+' ws term(w) «if type(v) = int = type(w) then v := v + w else error»
    | '|' ws term(w) «if type(v) = bool = type(w) then v := v or w else error» }
term(v) → factor(v)
    { '*' ws factor(w) «if type(v) = int = type(w) then v := v * w else error»
    | '&' ws factor(w) «if type(v) = bool = type(w) then v := v and w else error» }
factor(v) → integer(v) | '(' expression(v) ')' ws
integer(v) → digit(v) { digit(w) « v := 10 * v + w » } ws
digit(v) → '0' « v := 0 » | … | '9' « v := 9 »
ws → { ' ' }
```

More generally, attribute rules could allow conversion between types, e.g. from integer to floating point.

In the implementation, the type annotation now specifies that the value returned by `expression`, `simpleExpr`, `term`, `factor` is either `int` or `bool`:

In [ ]:
```
from typing import Union

pos: int; sym: str; src: str

def nxt():
    global pos, sym
    if pos < len(src): sym, pos = src[pos], pos + 1
    else: sym = chr(0) # end of input symbol

def expression() -> Union[bool, int]:
```

```python
    # expression → ws simpleExpression [ ( '=' | '<' ) ws simpleExpression]
    ws(); v = simpleExpr()
    if sym == '=':
        nxt(); ws(); w = simpleExpr()
        if type(v) == type(w): v = v == w
        else: raise Exception("incompatible operands of '=' at " + str(pos))
    elif sym == '<':
        nxt(); ws(); w = simpleExpr()
        if type(v) == int == type(w): v = v < w
        else: raise Exception("not int operands of '<' at " + str(pos))
    return v

def simpleExpr() -> Union[bool, int]:
    # simpleExpression → ws term { ('+' | '|' ) ws term }
    v = term()
    while sym in '+|':
        if sym == '+':
            nxt(); ws(); w = term()
            if type(v) == int == type(w): v = v + w
            else: raise Exception("not int operands of '+' at " + str(pos))
        else: # sym == '|'
            nxt(); ws(); w = term()
            if type(v) == bool == type(w): v = v or w
            else: raise Exception("not bool operands of '|' at " + str(pos))
    return v

def term() -> Union[bool, int]:
    # term → factor { ( '*' | '&' ) ws factor }
    v = factor()
    while sym in '*&':
        if sym == '*':
            nxt(); ws(); w = factor()
            if type(v) == int == type(w): v = v * w
            else: raise Exception("not int operands of '*' at " + str(pos))
        else: # sym == '&'
            nxt(); ws(); w = factor()
            if type(v) == bool == type(w): v = v and w
            else: raise Exception("not bool operands of '&' at " + str(pos))
    return v

def factor() -> Union[bool, int]:
    # factor → integer | '(' expression ')' ws
    if '0' <= sym <= '9': v = integer()
    elif sym == '(':
        nxt(); v = expression()
        if sym == ')': nxt(); ws()
        else: raise Exception("')' expected at " + str(pos))
    else: raise Exception("invalid character at " + str(pos))
    return v

def integer() -> int:
    # integer → digit { digit } ws
    # '0' <= sym <= '9'
    v = digit()
    while '0' <= sym <= '9': v = 10 * v + digit()
    ws()
    return v

def digit() -> int:
    # digit → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    # '0' <= sym <= '9'
    v = ord(sym) - ord('0'); nxt()
    return v

def ws(): # ws → { ' ' }
    while sym == ' ': nxt()

def evaluate(s: str) -> int:
    global src, pos;
    src, pos = s, 0; nxt(); v = expression()
    if sym != chr(0): raise Exception("unexpected characters at " + str(pos))
    return v
```

In [ ]:

```python
#evaluate("2 = (3 < 4)") # incompatible operands of '=' at 11
#evaluate("2 < (3 = 4)") # not int operands of '<' at 11
#evaluate("(2 = 3) + 4") # not int operands of '+' at 11
#evaluate("2 | 3") # not bool operands of '|' at 5
#evaluate("2 * (3 < 4)") # not int operands of '*' at 11
#evaluate("2 & 3") # not bool operands of '&' at 5
evaluate("(2 < 3) = (3 < 4)") # returns True
```

Infix to Postfix

In the *postfix* notation (or *RPN*, *Reverse Polish Notation*) for expressions, operators are written after the operands. Expressions are evaluated by first pushing the operands on a stack and when applying an operator, popping the operands from the stack and pushing the result on the stack again. Postfix notation does not need parentheses.

infix notation	postfix notation
`2 + 3`	`2 3 +`
`2 * 3 + 4`	`2 3 * 4 +`
`2 + 3 * 4`	`2 3 4 * +`
`(5 – 4) * (3 + 2)`	`5 4 – 3 2 + *`

Some dedicated programming languages like Postscript and Forth, a series of HP calculators, and some calculator apps use postfix notation. Some processors support evaluation of postfix expressions by providing a stack and instructions which operate this way on the stack ("zero address instructions"), in particular some CISC processors and virtual machines (JVM, .NET, WebAssembly).

In the attribute grammar below, all attributes are strings and `+` is used for concatenation:

```
expression(p) → ws term(p) { '+' ws term(q) « p := p + ' ' + q + '+' » }
term(p) → factor(p) { '*' ws factor(q) « p := p + ' ' + q + '*' » }
factor(p) → integer(p) | '(' expression(p) ')' ws
integer(p) → digit(p) { digit(q) « q := q + p» } ws
digit(p) → '0' « p := '0' » | ... | '9' « p := '9' »
ws → { ' ' }
```

The implementation follows the same principles as the previous one:

In [ ]:
```
pos: int; sym: str; src: str

def nxt():
    global pos, sym
    if pos < len(src): sym, pos = src[pos], pos + 1
    else: sym = chr(0) # end of input symbol

def expression() -> str: # expression → ws term { '+' ws term }
    ws(); p = term()
    while sym == '+': nxt(); ws(); p += ' ' + term() + ' +'
    return p

def term() -> str: # term → factor { '*' ws factor }
    p = factor()
    while sym == '*': nxt(); ws(); p += ' ' + factor() + ' *'
    return p

def factor() -> str: # factor → integer | '(' expression ')' ws
    if '0' <= sym <= '9': p = integer()
    elif sym == '(':
        nxt(); p = expression()
        if sym == ')': nxt(); ws()
        else: raise Exception("')' expected at " + str(pos))
    else: raise Exception("invalid character at " + str(pos))
    return p

def integer() -> str: # integer → digit { digit }
    p = digit()
    while '0' <= sym <= '9': p += digit()
    ws()
    return p

def digit() -> str: # digit → '0' | ... | '9'
    # '0' <= sym <= '9'
    p = sym; nxt()
    return p

def ws(): # ws → { ' ' }
    while sym == ' ': nxt()

def convert(s) -> str:
    global src, pos;
    src, pos = s, 0; nxt(); v = expression()
    if sym != chr(0): raise Exception("unexpected character at " + str(pos))
    return v
```

In [ ]:
```
convert("(2 + 3) * 4 + 5") # returns '2 3 + 4 * 5 +'
```

Abstract Syntax Tree

Consider the construction of an abstract syntax tree for arithmetic expressions over identifiers and operators for negation, addition, and multiplication. An abstact syntax tree is concisely captured by an inductive type definition,

```
Expr = Ident(str) | Minus(Expr) | Plus(Expr, Expr) | Times(Expr, Expr)
```

where `Ident`, `Minus`, `Plus`, `Times` are constructors. For example, the abstract syntax tree of `- (a + b)` is `Minus(Plus('a', 'b'))`. A grammar suitable for parsing is:

```
expression → ws term { '+' ws term }
term → factor { '*' ws factor }
factor → [ '-' ws ] atom
atom → identifier | '(' expression ')' ws
identfier → letter { letter } ws
letter(t) → 'a' | … | 'z'
ws → { ' ' }
```

(The choice of the word `atom` for something that is composed may seem odd for the time being.) The grammar is extended with attribute rules that construct the abstract syntax tree. The attribute `t` of `expression`, `term`, `factor`, and `atom` is of type `Expr`:

```
expression(t) → ws term(t) { '+' ws term(u) « t := Plus(t, u) » }
term(t) → factor(t) { '*' ws factor(u) « t := Times(t, u) » }
factor(t) →  '-' ws atom(t) « t := Minus(t) » | atom(t)
atom(t) → identifier(i) « t := Ident(i) » | '(' expression(t) ')' ws
identfier(i) → letter(i) { letter(l) « i := i + l » } ws
letter(l) → 'a' « l := 'a' » | … | 'z' « l := 'z' »
ws → { ' ' }
```

Note how the production of `factor` was reformulated without `[…]`.

In Python the type `Expr` can be expressed by a class and the constructors `Ident`, `Minus`, `Plus`, `Times` by subclasses of that class.

In [ ]:
```
from textwrap import indent

class Expr: pass

class Ident(Expr):
    def __init__(self, ident: str):
        self.ident = ident
    def __str__(self) -> str:
        return self.ident

class Minus(Expr):
    def __init__(self, arg: Expr):
        self.arg = arg
    def __str__(self) -> str:
        return '-\n' + indent(str(self.arg), '  ')

class Times(Expr):
    def __init__(self, left: Expr, right: Expr):
        self.left, self.right = left, right
    def __str__(self) -> str:
        return '*\n' + indent(str(self.left), '  ') + '\n' + indent(str(self.right), '  ')

class Plus(Expr):
    def __init__(self, left: Expr, right: Expr):
        self.left, self.right = left, right
    def __str__(self) -> str:
        return '+\n' + indent(str(self.left), '  ') + '\n' + indent(str(self.right), '  ')

pos: int; sym: str; src: str

def nxt():
    global pos, sym
    if pos < len(src): sym, pos = src[pos], pos + 1
    else: sym = chr(0) # end of input symbol

def expression() -> Expr: # expression → ws term { '+' ws term }
    ws(); t = term()
    while sym == '+': nxt(); ws(); t = Plus(t, term())
    return t

def term() -> Expr: # term → factor { '*' ws factor }
    t = factor()
    while sym == '*': nxt(); ws(); t = Times(t, term())
```

```
    return t

def factor() -> Expr: # factor → [ '-' ws ] atom
    if sym == '-': nxt(); ws(); t: Expr = Minus(atom())
    else: t = atom()
    return t

def atom() -> Expr: # atom → integer | '(' expression ')' ws
    if 'a' <= sym <= 'z': t: Expr = Ident(identifier())
    elif sym == '(':
        nxt(); t = expression()
        if sym == ')': nxt(); ws()
        else: raise Exception("')' expected at " + str(pos))
    else: raise Exception("invalid character at " + str(pos))
    return t

def identifier() -> str:
    # identfier(i) → letter(i) { letter(l) « i := i + l » } ws
    i = letter()
    while 'a' <= sym <= 'z': i += letter()
    ws()
    return i

def letter() -> str: # identfier → letter { letter } ws
    # 'a' <= sym <= 'z'
    l = sym; nxt()
    return l

def ws(): # ws → { ' ' }
    while sym == ' ': nxt()

def ast(s) -> Expr:
    global src, pos;
    src, pos = s, 0; nxt(); t = expression()
    if sym != chr(0): raise Exception("unexpected character at " + str(pos))
    return t
```

A method `__str__` is defined in `Expr` and each subclass to allow the abstract syntax tree to be printed sideways, for example:

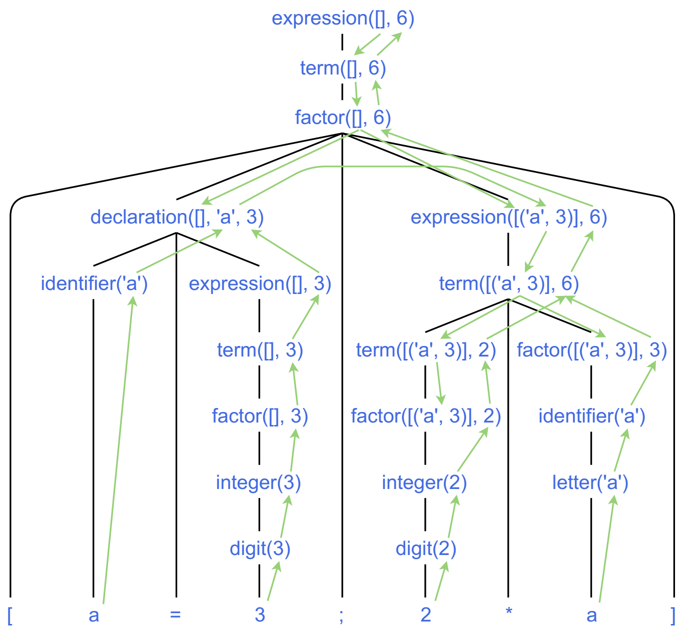
In [ ]:
```
print(ast('- (a + b) * c'))
```

Synthesized and Inherited Attributes

All attributes so far are called *synthesized* as they are computed "bottom-up": values are passed from the right-hand side of productions to the nonterminal on the left-hand side. The dual are called *inherited* attributes as they are computed "top-down": values are passed from the nonterminal on the left-hand side to the right-hand side. Both are written in parenthesis after the symbol.

As an example, consider extending adding constant declarations of the form `[a = 3; 2 * a]`, with the scope explicitly written in square brackets. A suitable grammar, without white space, is:

```
expression → term { '+' term }
term → factor { '*' factor }
factor → integer | identifier | '(' expression ')' |
    '[' declaration expression ']'
declaration → 'identifier '=' expression ';'
identifier → letter { letter }
letter → 'a' | … | 'z'
integer → digit { digit }
digit → '0' | … | '9'
```



Here, the declaration before `;` would synthesize the pair with `'a'` and `3`, which is then inherited when evaluating `2 * a`. Since there can be more than one declaration, pairs are kept in a list, which is used as a stack: each declaration adds a pair to the front of the list. In case of nested declarations, the same identifier for a constant can appear multiple times. The auxiliary function `lookup(d, i)` searches for the first occurrence of `i` in the list `d` of declarations:

```
expression(d, v) → term(d, v) { '+' term(d, w) « v := v + w » }
term(d, v) → factor(d, v) { '*' factor(d, f) « v := v × f }
factor(d, v) → integer(v) | identifier(i) « v ← lookup(d, i) » | '(' expression(d, v) ')' |
    '[' declaration(d, i, w) expression([(i, w)] + d, v) ']'
declaration(d, i, v) → identifier(i) '=' expression(d, v) ';'
```

```
identifier(i) → letter(i) { letter(l) « i := i + l » }
letter(l) → 'a' « l := 'a' » | … | 'z' « l := 'z' »
integer(i) → digit(i) { digit(d) « i := 10 × i + d » }
digit(d) → '0' « d := 0 » | … | '9' « d := 9 »
```

The start symbol is now `expression([], v)`; that is, initially the list of declarations is empty.

The rule for constructing parsing procedures is extended by having inherited attributes, `ai` below, and synthesized attributes, `as` below:

`p`	`pr(p)`
`B(ai, as) → E`	`procedure B(ai) → (as)` `pr(E)`

In Python, inherited attributes simply become parameters, e.g. for above grammar:

In [ ]:
```python
from typing import List, Tuple

pos: int; sym: str; src: str

def nxt():
    global pos, sym
    if pos < len(src): sym, pos = src[pos], pos + 1
    else: sym = chr(0) # end of input symbol

def lookup(d: List[Tuple[str, int]], i: str) -> int:
    for j, v in d:
        if i == j: return v
    raise Exception("undefined identifier at " + str(pos))

def expression(d: List[Tuple[str, int]]) -> int:
    # expression(d, v) → term(d, v) { '+' term(d, w) « v := v + w » }
    v = term(d)
    while sym == '+': nxt(); w = term(d); v = v + w
    return v

def term(d: List[Tuple[str, int]]) -> int:
    # term(d, v) → factor(d, v) { '*' factor(d, f) « v := v * f }
    v = factor(d)
    while sym == '*': nxt(); w = factor(d); v = v * w
    return v

def factor(d: List[Tuple[str, int]]) -> int:
    # factor(d, v) → integer(v) | identifier(i) « v ← lookup(d, i) » | '(' expression(d, v) ')' |
    #     '[' declaration(d, i, w) expression([(i, w)] + d, v) ']'
    if '0' <= sym <= '9': v = integer()
    elif 'a' <= sym <= 'z': i = identifier(); v = lookup(d, i)
    elif sym == '(':
        nxt(); v = expression(d)
        if sym == ')': nxt()
        else: raise Exception("')' expected at " + str(pos))
    elif sym == '[':
        nxt(); i, w = declaration(d)
        v = expression([(i, w)] + d)
        if sym == ']': nxt()
    else: raise Exception("invalid character at " + str(pos))
    return v

def declaration(d: List[Tuple[str, int]]) -> Tuple[str, int]:
    # declaration(d, i, v) → identifier(i) '=' expression(d, v) ';'
    i = identifier()
    if sym == '=': nxt()
    else: raise Exception("'=' expected at " + str(pos))
    v = expression(d)
    if sym == ';': nxt()
    else: raise Exception("';' expected at " + str(pos))
    return (i, v)

def identifier() -> str:
    # identifier(i) → letter(i) { letter(l) « i := i + l » }
    i = letter()
    while 'a' <= sym <= 'z': i = i + letter()
    return i

def letter() -> str:
    # letter(l) → 'a' « l := 'a' » | … | 'z' « l := 'z' »
    if 'a' <= sym <= 'z': l = sym; nxt()
    else: raise Exception("letter expected at " + str(pos))
    return l
```

```
def integer() -> int:
    # integer(i) → digit(i) { digit(d) « i := 10 × i + d » }
    # '0' <= sym <= '9'
    i = digit()
    while '0' <= sym <= '9': i = 10 * i + digit()
    return i

def digit() -> int:
    # digit(d) → '0' « d := 0 » | … | '9' « d := 9 »
    # '0' <= sym <= '9'
    d = ord(sym) - ord('0'); nxt()
    return d

def evaluate(s):
    global src, pos;
    src, pos = s, 0; nxt(); v = expression([])
    if sym != chr(0): raise Exception("unexpected character at " + str(pos))
    return v
```

In [ ]:

```
#evaluate("[a=3;a]+a") # undefined identifier at 14
#evaluate("[a+2]") # '=' expected at 4
#evaluate("[a=2]") # ';' expected at 7
#evaluate("[1=2]") # letter expected at 2
(evaluate("(2+[pi=3;2*pi])*2"), # returns 16
 evaluate("(2+[pi=3;[pi=1;pi*2]*pi])*2"), # returns 16
 evaluate("[a=2;[a=a+1;a]]")) # returns 3
```

Note that the result of `3` for the last example is indeed correct: the inner declaration of `a` evaluates the `a + 1` with the outer value of `a`.

Historic Notes and Further Reading

Attribute grammars were suggested by Donald Knuth for assigning semantics to context-free languages (Knuth 1968); he also provides a fascinating recollection of the origins (Knuth 1990). In the common definition of an attribute grammar, attributes are functions over symbols, so `a(X)`, `b(X)` refer to attributes `a`, `b` of `X`. We write instead `X(a, b)` for brevity and because of the close correspondence to an implementation in a programming language.

The original proposal allows for arbitrary dependencies among synthesized and inherited attributes. As this may lead to circular dependencies, a check for well-formedness is necessary. Consider a grammar in which nonterminal `A` is defined by a set of productions of the form

$$A \to X_1\ X_2\ \ldots$$

where each X_i is a terminal or nonterminal. If each inherited attribute of X_i depends only on attributes of $X_1, \ldots, X_{i-1}$ and the inherited attributes of A, then the attribute grammar is called *L-attributed*. This condition guarantees the absence of a circular dependency of attributes. In implementation terms, the value parameters of parsing procedure X_i have to depend only the parameters of $X_1, \ldots, X_{i-1}$ and the value parameters of `A`. The attribute grammars in these notes are all L-attributed. This allows the attribute rules to be directly embedded in the parsing procedures and there is no need to build the parse tree as a data structure in which nodes have to be revisited. A classification of attribute grammars is given in (Paakki 1995).

The example of type-checking shows how attribute grammars can express context-dependencies, which could in principle be achieved with context-sensitive grammars. However, since the computation of attributes can involve arbitrary functions, anything computable can be expressed with attribute grammars; hence, they have the same expressiveness as unrestricted grammars, which are equivalent to Turing machines.

Since the conception of attribute grammars, numerous tools have been developed that take an attribute grammar as input and generate a parser for a specific language, essentially automating the manual implementation of the previous examples. Such tools are known for generating less efficient parsers than hand-written ones; from a practical point, a drawback is the dependence on such a tool. An intriguing alternative is to express the whole attribute grammar as an executable (functional) program (Viera et al. 2009); we leave it to the reader to judge the simplicity and efficiency of that approach.

Exercises

Exercise 1. Given above plain grammar for binary numbers, what is an attribute grammar for computing the number of zero's and one's of sequence of digits? Draw the parse tree for `1011` and annotate each node with the attribute values! *Hint:* Use two attributes, one for the number of zero's and one for the number of one's.

production	**attribute rule**
$\text{binary}(z_0, o_0) \to \text{binary}(z_1, o_1)\ \text{digit}(z_2, o_2)$	$z_0, o_0 := z_1 + z_2, o_1 + o_2$
$\text{binary}(z_0, o_0) \to \text{digit}(z_1, o_1)$	$z_0, o_0 := z_1, o_1$
$\text{digit}(z, o) \to$ '0'	$z, o := 1, 0$

`digit(z, o) → '1'`	`z, o := 0, 1`

Bibliography

Knuth, Donald E. 1968. "Semantics of Context-Free Languages." *Mathematical Systems Theory* 2 (2): 127–45. https://doi.org/10.1007/BF01692511.

———. 1990. "The Genesis of Attribute Grammars." In *Attribute Grammars and Their Applications*, edited by P. Deransart and M. Jourdan, 1–12. Lecture Notes in Computer Science. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-53101-7_1.

Paakki, Jukka. 1995. "Attribute Grammar Paradigms—a High-Level Methodology in Language Implementation." *ACM Computing Surveys* 27 (2): 196–255. https://doi.org/10.1145/210376.197409.

Viera, Marcos, S. Doaitse Swierstra, and Wouter Swierstra. 2009. "Attribute Grammars Fly First-Class: How to Do Aspect Oriented Programming in Haskell." In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, 245–56. ICFP '09. New York, NY, USA: ACM. https://doi.org/10.1145/1596550.1596586.