Item 1: Syntax-directed translation

Mari Paz Guerrero Lebrero

Grado en Ingeniería Informática

Curso 2018/2019

Semantic

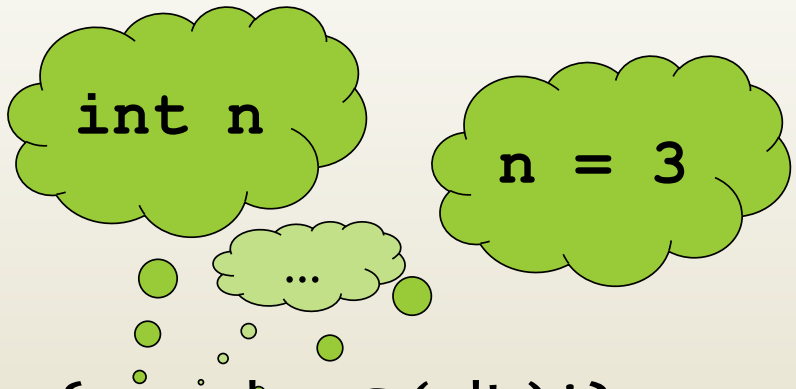
- The compiler should keep invariant meaning of the program, regardless of the language syntax in which it is written
- The semantics informally is described in natural language for most programming languages ; two compilers might give different results for the same program source
- Unlike what happens with the syntax, there is not a unified semantics of programming languages regarding treatment:
  - Semantics concept of a programming language
  - Notations to describe the semantics
  - Methods to build translators from the semantic description of a language

Semantic formalization

- ***Operational semantics*:** The semantics of the programming language is described in an abstract virtual machine
- **Denotational semantics:** Each primitive syntactic object is mapped to an object of an abstract space of primitive meanings (space denotations); each operator of language is mapped to a function in the space denotations
- **Axiomatic semantics:** By logical formulas that must be met before (preconditions) and after (post-conditions) of defined properties must meet each syntactic structure, and therefore the program.

Semantic Analysis phase (Contextual constraints)

- Not all constructions of programming languages can be expressed by Context-free grammars
- Context free grammar can not express the problem of checking the declaration of variables before use



$$L_1 = \{wcw \mid w \in (a|b)^*\}$$

Attributed grammars

- Associate ATTRIBUTES (information) to the grammar symbols.
- A symbol can have one, none, or many different attributes.
- The notation we use for attributes is the same as for the properties of objects.
- In a way there is a similarity between:
  - grammar symbol ≈ class
  - instance of a symbol ≈ object
  - Attribute of a symbol ≈ object property
  - All instances of the symbol have the same attributes but their actual values may differ.
  - To distinguish between different instances of the same symbol used subindexes.
- A syntactic rules we add 'semantic' rules indicating how these attributes are handled.

Example: a calculator

Sintactic rules	Semantics rules
entrada → entrada expr ';'	Escribe(expr.s)
entrada → ξ	
expr → $expr_1$ '+' sum	expr.s := $expr_1$.s + sum.s
expr → sum	expr.s := sum.s
sum → sum_1 '*' factor	sum.s := sum_1.s * factor.s
sum → factor	sum.s := factor.s
factor → '-' $factor_1$	factor.s := - $factor_1$.s
factor → NUM	factor.s := NUM.lexval
factor → '(' expr ')'	factor.s := expr.s

Symbol	Attribute
expr	expr.s
sum	sum.s
factor	factor.s
NUM	NUM.lexval

Attributes intend to be the value of that support analysis hive

To all this we call a syntax-directed definition

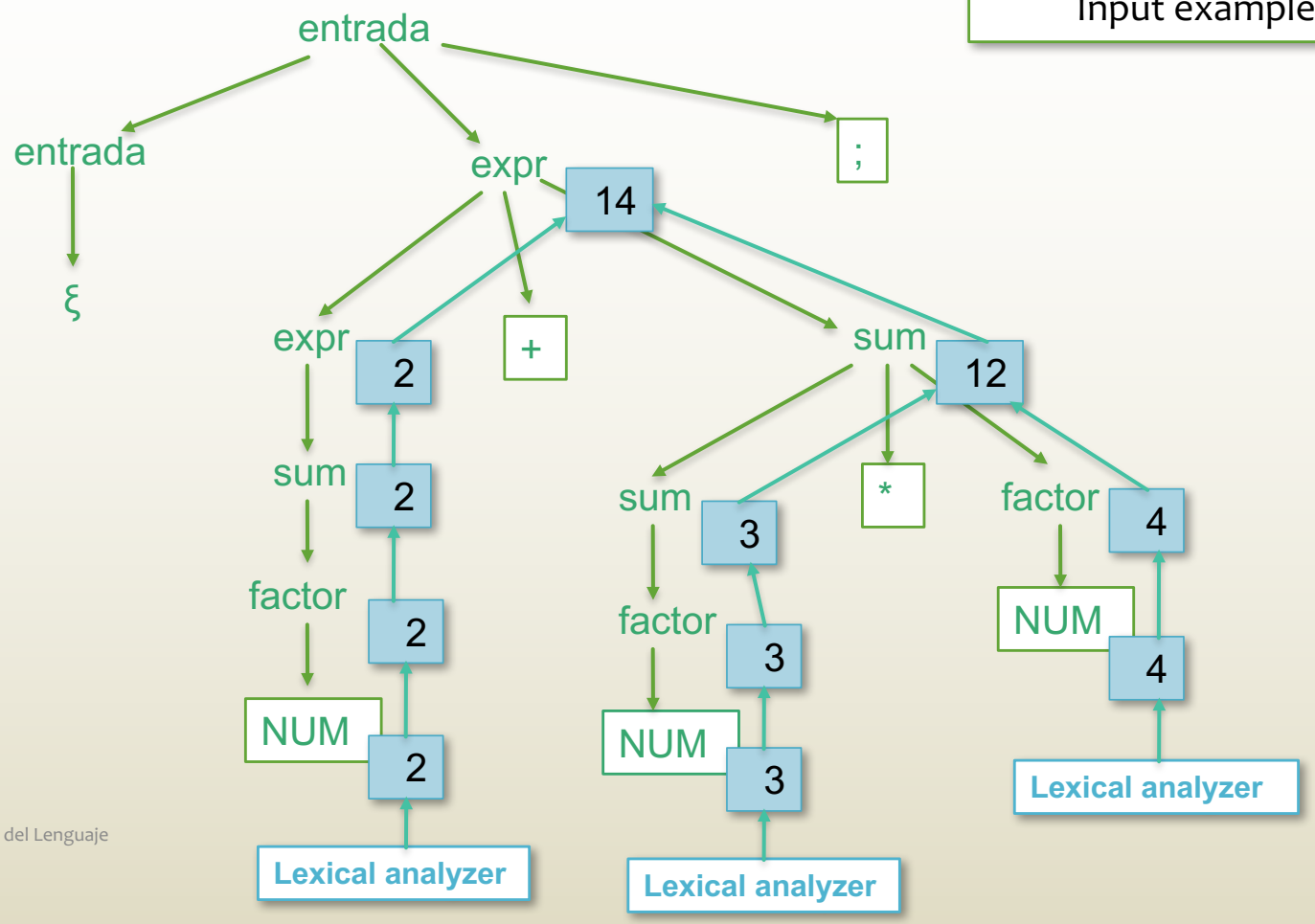
Decorated tree example

Input example: 2+3*4;

In parallel to the analysis tree

We're building a graph with attributes.

The arrow shows how information flows (the agency that indicate semantic rules)



Inductive Reasoning

- To understand the above syntax directed definition is convenient reason inductively:
  - The base cases are resolved by hand (lexical tokens values are assigned by the lexical analyzer).
  - For the remaining cases it must be assumed that a certain "induction hypothesis' for subcases are met and then solve the most complicated cases.
  - The induction hypothesis:
    - **$expr \rightarrow expr_1 + sum \{expr.s := expr_1.s + sum.s\}$**
    - It is that **$expr_1.s$** has the value of the underlying hive, while **sum.s** has the value (this is supposed already solved) and under that assumption, the value corresponding to **expr.s** tree would the total value of the hives.

Ascending implementation (SLY)

```
@_('entrada expr ";"')
def entrada(self, p):
   print(p.expr)
@_(' ')
def entrada(self, p):
   print(" ")
@_('expr "+" sum')
def expr(self, p):
   return p.expr + p.sum
@_('sum')
def expr(self, p):
   return p.sum
@_('sum "*" factor')
def sum(self, p):
   return p.sum * p.factor
```

```
@_('factor')
def sum(self, p):
   return p.factor
@_('"-" factor')
def factor(self, p):
   return -p.factor
@_('NUMBER')
def factor(self, p):
   return p.NUMBER
@_('"(" expr ")"')
def factor(self, p):
   return p.expr
```

Sintactic rules	Semantics rules
entrada → entrada expr ';'	Escribe(expr.s)
entrada → ξ	
expr → $expr_1$ '+' sum	expr.s := $expr_1$.s + sum.s
expr → sum	expr.s := sum.s
sum → sum_1 '*' factor	sum.s := sum_1.s * factor.s
sum → factor	sum.s := factor.s
factor → '-' $factor_1$	factor.s := - $factor_1$.s
factor → NUM	factor.s := NUM.lexval
factor → '(' expr ')'	factor.s := expr.s

Sintact analyzer using SLY

Syntax-directed definition

- It is a grammar **G = {T, N, S, P}**
- A set of attributes to the grammar symbols divided into:
  - **synthesized** attributes
  - **inherited** attributes
- Each syntax rule: **A → α**, it is associated with a set of 'semantic' rules of the form: **b: = f (c1, c2, ..., ck)**
- Where **b** is a **synthesized** attribute of **A** and **c1, c2, ..., ck** are attributes of the symbols of **A** and / or **α**
- Or, **b** is **inherited** from one of the symbols **α** and attribute **c1, c2, ..., ck** are attributes of the symbols of **A** and / or **α**
- While **f** is any function or operation with these attributes

Synthesized vs. inherited

- **Synthesized** attributes are the parent attributes and they are calculated based on the children attributes (arrows flow upward graph).
- **Inherited** attributes are the children attributes that are calculated from parent and / or brothers attributes (the arrows flow down the graph and / or horizontal)
- The tokens are only **synthesized** attributes (assigned by the lexical analyzer)

Another example: binary numbers

Sintactic rules	Semantics rules
num → sec_1 '.' sec_2	num.v := sec_1.v + sec_2.v/2^sec_2.lon
sec → sec_1 dig	sec.v := sec_1.v * 2 + dig.v
	sec.lon := sec_1.lon + 1
sec → dig	sec.v := dig.v
	sec.lon := 1
dig → '0'	dig.v := 0
dig → '1'	dig.v := 1

Used attributes	
num.v	Number value
sec.v	Digit sequence value
sec.lon	Digit sequence length
dig.v	Digit value

This definition supports binary floating point numbers and calculates its value. Example: 10011.101

Ascending implementation (SLY)

```
from sly import Lexer, Parserclass
C
```

```
class CalcParser(Parser):
    tokens = CalcLexer.tokens
    def __init__(self):
        self.names = { }
    @_('num')
    def entrada(self, p):
      print("El valor es ", p.num)

    @_('sec "." sec')
    def num(self, p):
        return p.sec0[0] + p.sec1[0] / pow(2,p.sec1[1])

    @_('sec dig')
    def sec(self, p):
        aux = [p.sec[0] * 2 + p.dig, p.sec[1] + 1]
        return aux
    @_('dig')
    def sec(self, p):
        return [p.dig, 1]
    @_('DIGIT')
    def dig(self, p):
        return p.DIGIT
```

Sintactic rules	Semantics rules
num → sec_1 '.' sec_2	num.v := sec_1.v + sec_2.v/2^sec_2.lon
sec → sec_1 dig	sec.v := sec_1.v * 2 + dig.v
	sec.lon := sec_1.lon + 1
sec → dig	sec.v := dig.v
	sec.lon := 1
dig → '0'	dig.v := 0
dig → '1'	dig.v := 1

```
if__name__ == '__main__':
    lexer = CalcLexer()
    parser = CalcParser()
    while True:
        try:
            text = input('binary number > ')
        except EOFError:
            break
        if text:
            parser.parse(lexer.tokenize(text))
```

Example with inherited attributes

```
definicion -> tipo {lista.h := tipo.s;} lista ';'
tipo -> INTEGER {tipo.s := new NodoEntero;}
      | FLOAT {tipo.s := new NodoFlotante;}
      | CHAR {tipo.s := new NodoChar;}
lista -> {elm.h := lista.h;} elm {resto.h := lista.h;} resto
resto -> ',' {elm.h := resto.h;} elm {resto1.h := resto.h;} resto1
       |ξ
elm -> '*' {elm1.h := new NodPuntero(elm.h);} elm1
     | ID {tabla[ID.lexval] = elm.h;}
```

Synthesized attributes	Inherited attributes
tipo.s	lista.h
ID.lexval	elm.h
	resto.h

- Inherited attributes used when the value depends on the context
- The synthesized attributes are used when the value of father is obtained from the value of children

Ascending implementation (SLY)

```
class CalcLexer(Lexer):
    tokens = {ID, CHAR, INTEGER, FLOAT}
    ignore = ' \t'
    literals = { '*', ',', ';'}

# Regular expression rules for tokens
    ID             = r'[a-zA-Z_][a-zA-Z0-9_]*'
    ID['char']     = CHAR
    ID['integer'] = INTEGER
    ID['float']    = FLOAT

@_(r'\n+')
def newline(self, t):
    self.lineno += t.value.
def error(self, t):
    print("Illegal characte
    self.index += 1
```

```
class CalcParser(Parser):
    tokens = CalcLexer.tokens
    def __init__(self):
        self.names = { }

    @_('defi entrada')
    def entrada(self, p):
        pass

    @_(' ')
    def entrada(self, p):
        pass

    @_('tipo lista ";"')
    def defi(self, p):
        return p.tipo
```

```
    @_('INTEGER')
    def tipo(self, p):
        return NodoEntero()

    @_('FLOAT')
    def tipo(self, p):
        return NodoFlotante()

    @_('CHAR')
    def tipo(self, p):
        return NodoCaracter()

    @_(' empty5 elm empty4 resto')
    def lista(self, p):
        return p[-2]

    @_('')
    def empty4(self,p):
        return p[-3]
    @_('')
    def empty5(self,p):
        return p[-1]
```

Ascending implementation (SLY)

```
    @_('","  empty3 elm empty2 resto')
    def resto(self, p):
        return p[-4]
    @_('')
    def empty2(self,p):
        return p[-4]

    @_('')
    def empty3(self,p):
        return p[-2]

    @_('')
    def resto(self, p):
        return p[-1]

    @_('"*" empty1 elm')
    def elm(self, p):
        return p.empty1
    @_('')
    def empty1(self,p):
        return NodoPuntero(p[-2])
    @_('ID')
    def elm(self, p):
        tabla[p.ID] = p[-2].escribir()
```

```
if __name__ == '__main__':
    lexer = CalcLexer()
    parser = CalcParser()

    while True:
        try:
            text = input('data type list > ')

        except EOFError:
            break
        if text:
            parser.parse(lexer.tokenize(text))
        print(tabla)
```

Ascending implementation

```
tabla  = {}

class Nodo():
    def escribir(self):
        pass

class NodoEntero(Nodo):
    def escribir(self):
        return "Entero"

class NodoFlotante(Nodo):
    def escribir(self):
        return "Flotante"

class NodoCaracter(Nodo):
    def escribir(self):
        return "Caracter”

class NodoPuntero(Nodo):
    def __init__(self,n):
        self.n = n
    def escribir(self):
        return "Puntero a " + self.n.escribir()};
```

Down recursive implementation

- ***First set*** calculation:
  1. Initial case: If A -> TOKEN then FIRST(A) = {TOKEN}
  2. Initial case: If A -> ξ , then FIRST(A) = {ξ}
  3. Inductive case: If A -> B, then FISRT(A) = FIRST(B)
- ***Follow set*** calculation:
  1. The ***follow set*** of initial symbol is $ (EOF)
  2. If A -> α B β, then FOLLOW(B) = FIRST(β)
  3. If A -> α B $\vee$ A -> α B β $\wedge$ FIRST(β) $\subset$ ξ, then FOLLOW(B) = FOLLOW(A)

Down recursive implementation

FIRST SETS:

FIRST(elm) = {'*', ID}
FIRST(resto) = {',', ξ}
FIRST(lista) = FIRST(elm) = {'*', ID}
FIRST(tipo) = {INTEGER, FLOAT, CHAR}
FIRST(def) = FIRST(tipo) = {INTEGER, FLOAT, CHAR}
FIRST(ent) = FIRST(def) ∪ ξ= {INTEGER, FLOAT, CHAR, ξ}

FOLLOW SETS:

FOLLOW(ent) = {$}
FOLLW(def) = FIRST(ent) ∪ FOLLOW(ent) = {INTEGER, FLOAT, CHAR, $}
FOLLOW(tipo) = FIRST(lista) = {'*', ID}
FOLLOW(lista) = {';'}
FOLLOW(resto) = FOLLOW(lista) = {';'}
FOLLOW(elm) = FIRST(resto) ∪ FOLLOW(lista) = {',', ';'}

Down recursive implementation

```
from sly import Lexer
tabla  = {}
ta = None
ind = 0
tokens = None
tokenlist = None

class Nodo():
    def escribir():
        pass
class NodoEntero(Nodo):
    def escribir():
        return "Entero"
class NodoFlotante(Nodo):
    def escribir():
        return "Flotante"
class NodoCaracter(Nodo):
    def escribir():
        return "Caracter”
class NodoPuntero(Nodo):
    def __init__(self,n):
        self.n = n
    def escribir(self):
        return "Puntero a " + self.n.escribir()
```

- Synthesized attributes are the result of the function
- Inherited attributes are the function parameters

Synthesized attributes	Inherited attributes
tipo.s	lista.h
ID.lexval	elm.h
	resto.h

Down recursive implementation

```
def definicion():
    global ta, tokens
    if ta.type == tokens[0] or ta.type == tokens[1] or ta.type == tokens[3]:
        tipo_s = tipo()
        lista_h = tipo_s
        lista(lista_h)
        cuadra(';');
    else:
        yyerror("en definicion");
#fin de definicion()

def entrada():
    global ta, ind, tokens
    if ta.type == tokens[0] or ta.type == tokens[1] or ta.type == tokens[3]:
        definicion()
        entrada()
    else:
        if ind < len(tokenlist):  #fin de la entrada
            yyerror("en entrada")
#fin de entrada()
```

Entrada → definicion entrada
| ξ

definicion → tipo {lista.h := tipo.s;} lista ';'

Down recursive implementation

```
def lista(lista_h):
    global ta, tokens
    if ta.value == '*' or ta.type == tokens[2]:
        elm(lista_h);
        resto(lista_h);
#fin de lista()

def tipo():
    global ta, tokens
    if ta.type == tokens[3]:
        cuadra("INTEGER")
        return NodoEntero
    else:
        if ta.type == tokens[1]:
            cuadra("FLOAT")
            return NodoFlotante
        else:
            if ta.type == tokens[0]:
                cuadra("CHAR")
                return NodoCaracter
            else:
                yyerror("en tipo");
#fin de tipo()
```

```
tipo → INTEGER {tipo.s := new NodoEntero;}
     | FLOAT   {tipo.s := new NodoFlotante;}
     | CHAR    {tipo.s := new NodoChar;}

lista -> {elm.h := lista.h;} elm {resto.h := lista.h;} resto
```

Down recursive implementation

```
def elm(elm_h):
    global ta, tokens
    if ta.value == '*':
        cuadra('*')
        elm1_h = NodoPuntero(elm_h)
        elm(elm1_h)
    else:
        if ta.type == tokens[2]:
            IDlexval = ta.value
            cuadra("ID")
            tabla[IDlexval] = elm_h.escribir()
        else:
            yyerror("en elm")
#fin de elm()

def resto(resto_h):
    global ta
    if ta.value == ',':
        cuadra(',');
        elm(resto_h);
        resto(resto_h);
    else:
        if ta.value != ';':
            yyerror("en resto");
#fin de resto()
```

```
resto -> ',' {elm.h := resto.h;} elm {resto1.h := resto.h;} resto1
       | ξ
elm -> '*' {elm1.h := new NodPuntero(elm.h);} elm1
     | ID {tabla[ID.lexval] = elm.h;}
```

Down recursive implementation

```
def yyerror(msj):
    print("Error sistactico", msj)
#fin de yyerror()

def cuadra(obj):
    global ta, ind, tokenlist
    if ta.type == obj:
        #print("cuadro ta = ", ta.value)
        ind += 1
        if ind < len(tokenlist):
            ta = tokenlist[ind]
            #print("==> nuevo ta = ", ta.value)
    else:
        yyerror("en cuadra");
#fin de cuadra()
```

Down recursive implementation

```
if __name__ == '__main__':
    global ta, tokens
    lexer = CalcLexer()
    tokens = CalcLexer.tokens
    tokens = sorted(tokens)
    print("Programa que lee definiciones de variables\n")
    while True:
        try:
            data = input('data type list > ')
        except EOFError:
            break
        if data:
            tokenlist = list(lexer.tokenize(data))
            ta = tokenlist[ind]
            entrada()
        ind = 0
        print("Final de la entrada estas son las variables\n")
        print(tabla)
```

A implementation problem …

- Not all Syntax directed definitions can be implemented easily.
  - It may happen that the dependencies between attributes are not consistent with the order in which the parsing is done.
  - Consider a graph with circular arrows, from right to left etc.
- The definitions consistent with the LL(1) and LR(1) algorithms are called *L-Attributed Definitions* (L comes from Left).
- Definitions that have only synthesized attributes are called *S-Attributed* and these are a subset of the *L-Attributed*.
- The solution for a definition that is not *L-Attributed* would be:
  - Or find another solution
  - Or build an intermediate syntax tree (AST)

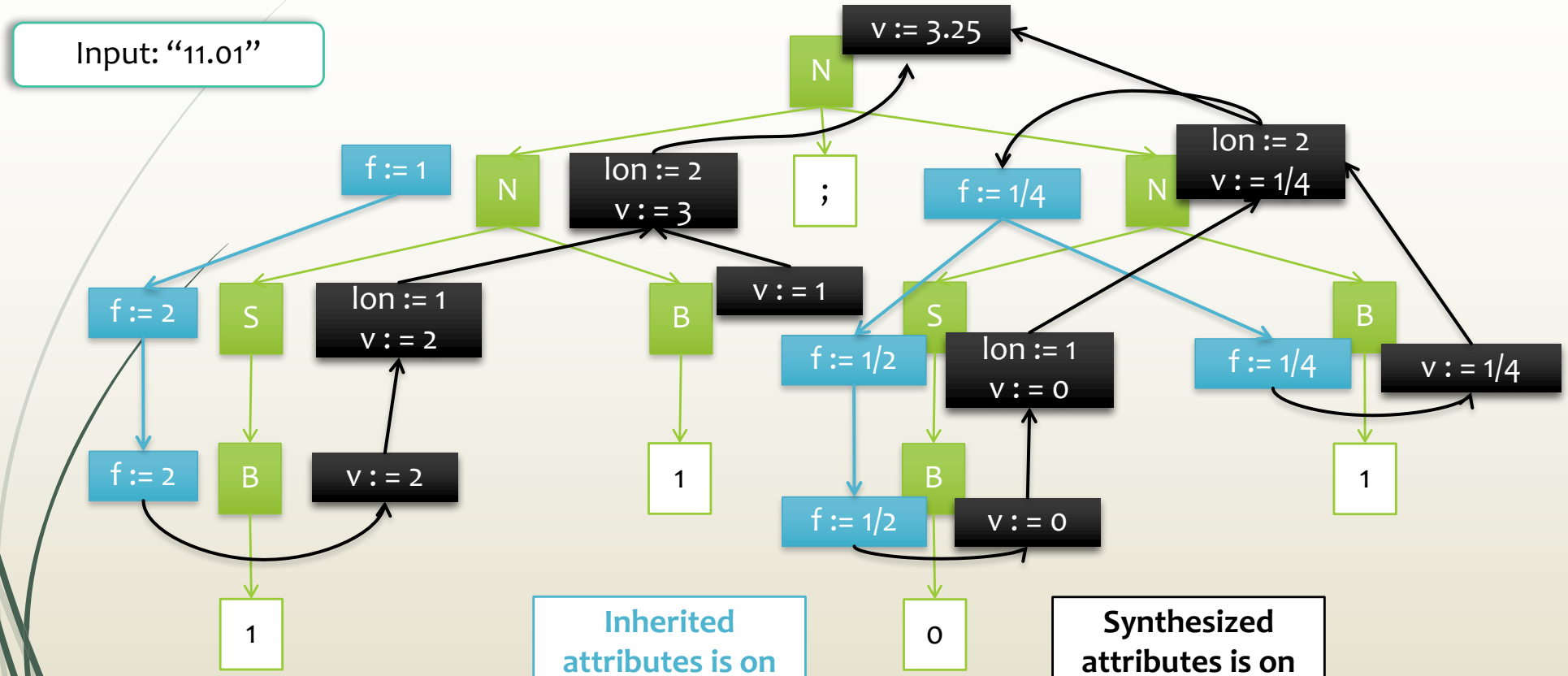
Definition that is not L-Attribute

Productions	Semantic rules
$N \rightarrow S_1$ '.' S_2	N. v := S_1.v + S_2.v S_1.f := 1 S_2.f := 2^ - S_2.lon
$S_0 \rightarrow S_1$ B	S_0.v := S_1.v + B.v S_1.f := 2 * S_0.f B.f := S_0.f S_0.lon := S_1.lon + 1
S→B	B.f := S.f S.v := B.v S.lon := 1
B → '0'	B.v := 0
B → '1'	B.v := B.f

Inherited attributes	
B.f	Power of two, value of each position
S.f	Power of two of fist digit sequence

Synthesized attributes	
N.v	Number value
B.v	Digit value
S.v	Digit sequence value
S.lon	Digit sequence length

Decorated tree analysis



L – attributed definition

- An inherited attribute of a child can only depend on inherited attributes of the father and the (inherited and / or synthesized) attributes of the brothers who are on the left.
- This definition is L – attributed

Syntactic rule	Semantic rules
A → B C D	A.s := B.s + D.h C.h := A.h + B.s

- This definition is not L - attributed

Syntactic rule	Semantic rules
A → B C D	A.s := B.s + D.h C.h := A.s + D.s

Translation schemes

- Once we have assured that definition is L-Attributed, we must build a translation scheme.
- A translation scheme is similar to a definition but the rules semantic (semantic actions) are enclosed in braces and tucked into the body of rules.
- This indicates when shoot semantic actions: by the time they reach parsing, if we consider the actions as if they were a symbol.

Constrains to build a translation scheme

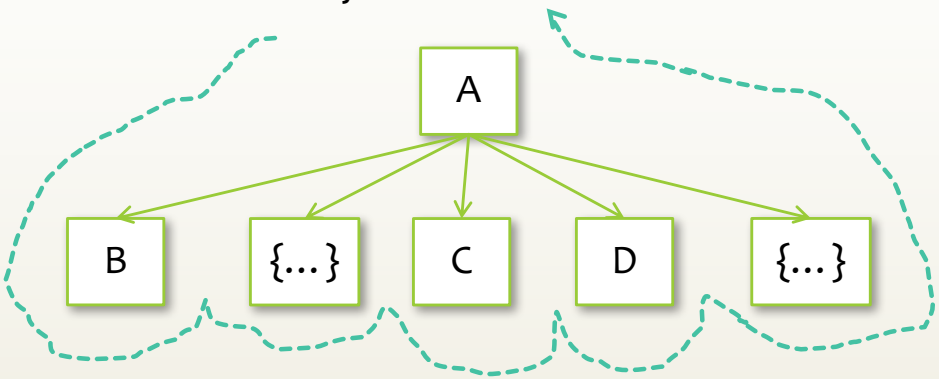
- Measures using synthesized attributes must be placed somewhere after the symbol
  - To remember this limitation: the descending implementation, the synthesized attributes of certain symbol are the result of the function corresponding to the symbol; so they can only be used after calls to that function.
- Evaluating actions (allocated) inherited attributes of a symbol, they must be placed before the symbol (usually immediately before).
  - To remember this limitation: in the downward deployment, they inherited attributes of a certain symbol are the parameters that will be passed to the function corresponding to the symbol; so we have to be calculated before calling this function.
- The action that evaluates the synthesized attribute header can (and usually) get to the end of that rule.
- The tokens are only synthesized attributes and are calculated by the lexical analyser.

Example: schema definition step

Syntax-directed definition

Sytactic rule	Semantic rules
`A → B C D`	`A.s := B.s + D.h` `C.h := A.h + B.s`

Syntax tree



Translation scheme

```
A → B {C.h:= A.h + B.s} C D {A.s:= B.s + D.h}
```

Translation scheme ascending implementation (SLY)

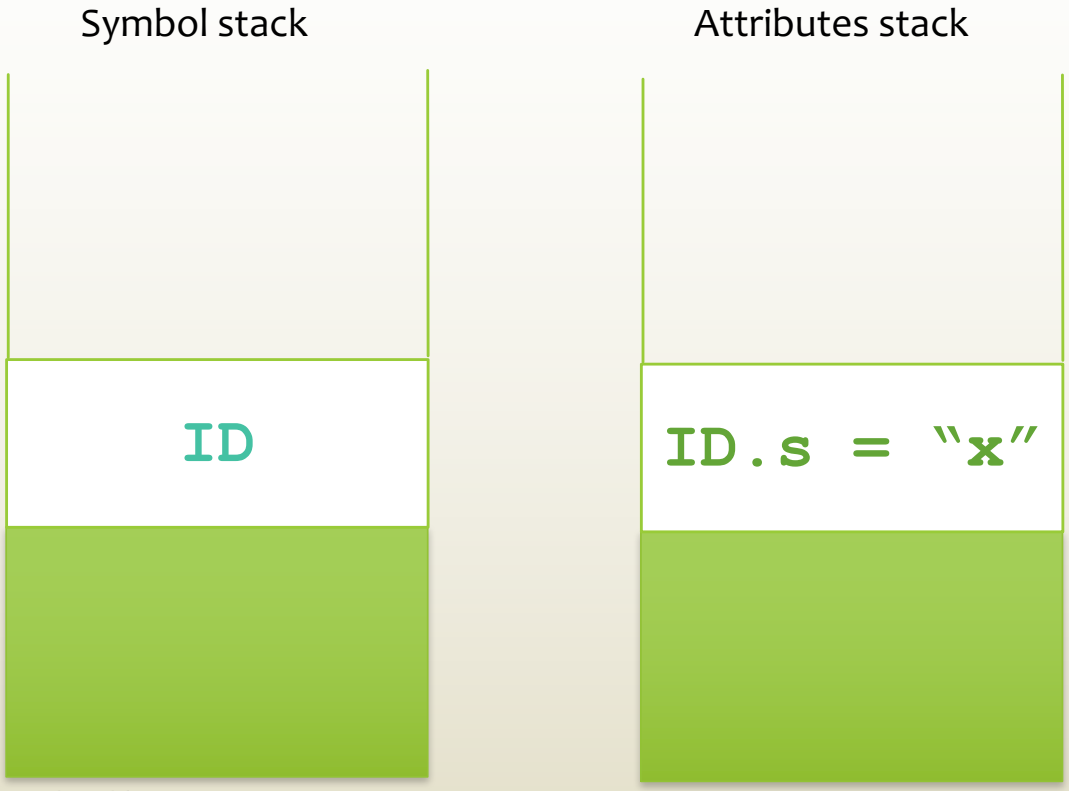
- Attributes stack is used in parallel with the symbols stack(actually, a stack of states).
- Lexical analyzer must obtain the lexical value (synthesized attribute tokens) and then return the token:

```
tokens = {ID}
ID = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

Actions during a shift

- The algorithm works by SLY shift / reduce:
  - During a shift, a token (actually a state) is get on the symbols stack.
  - In parallel, during shift, the value of the *yylval* variable (the attribute of the token) is get on the attributes stack.

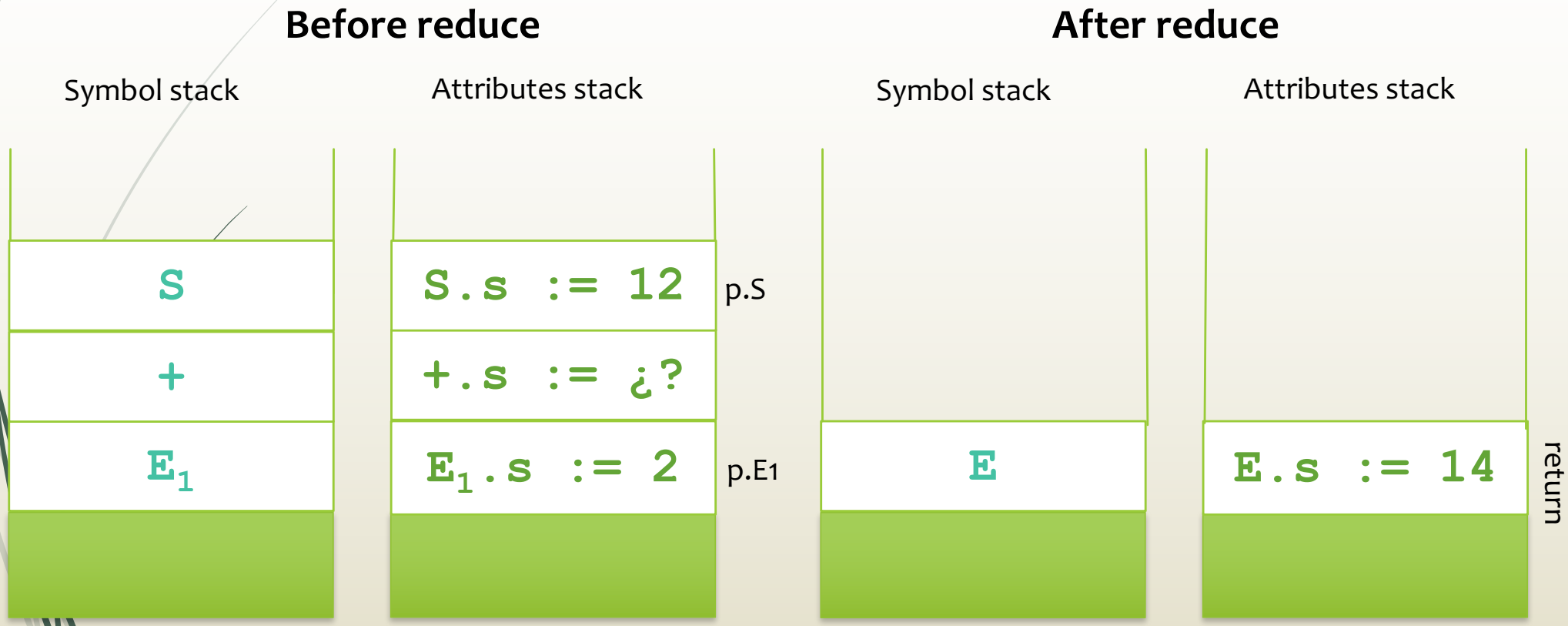
Actions during a shift



Actions during a reduce

- For a reduces by rule **A → α**, symbols of symbols stack (as many as α indicate the body) are removed and replaced by his bedside **A**.
  - For example: E → E1 '+' S {E.s: = E1.s + S.S}.
  - Here, 3 symbols of symbols stack would be removed and would get the E.
- In parallel, the semantic action is triggered by removing the attributes corresponding to the body of the attributes stack and replacing the (synthesized) attribute by header attribute.
  - The above example in SLY: E: E1 '+' S -> return p.E1 + p.S
  - Semantic action *return p.E1 + p.S* is triggered during reduced.
  - *return* indicates synthesized attribute of the header (would be placed on the attribute stack after reduction)
  - p.E1 and p.S represent the synthesized attributes of the 1st and 3rd symbol, placed on the stack of attributes relative depth 1 and 3 (before reduction, would then deleted).

Actions during a reduce



Some tricks to consider…

- In fact, only synthesized attributes exist in the attributes stack . So how do you handle / simulate inherited attributes?
  - Answer: to access attributes previously placed on the stack at a certain depth known.
  - Semantic shares placed at the end of a rule soar with reduced that rule.
  - Semantic shares placed within a rule (not the end), SLY performs automatically a transformation inventing a new symbol and a new rule, so that just shooting with reduced.
- Example: A: X {print ("hello");} Y Z
  - It automatically become the following: A → X @1 Y Z
  - @1 → ξ {print ("hello");}
  - Being @1 a new symbol generated by SLY, and so that the semantic action ends at the end of a rule is triggered when the rule is reduced

Example using SLY

Input	Output
	`# comienza el programa`

`// esto es un comentario`
`int a, b[3], *c;`
	`T0 = entero,`
	`asserta(tipo('a',T0)),`
	`T1 = array(3,T0),`
	`asserta(tipo('b',T1)),`
	`T2 = puntero(T0),`
	`asserta(tipo('c',T2)),`

`char **hola[2][3], adios; // hay que`
`// seguir filtrando`
	`T3 = caracter,`
	`T4 = puntero(T3),`
	`T5 = puntero(T4),`
	`T6 = array(3,T5),`
	`T7 = array(2,T6), asserta(tipo('hola',T7)),`
	`asserta(tipo('adios',T3)),`

`float *m, **d, c[34][56]; // adios esto es`
`otro comentario`
	`T8 = flotante,`
	`T9 = puntero(T8),`
	`asserta(tipo('m',T9)),`
	`T10 = puntero(T8),`
	`T11 = puntero(T10),`
	`asserta(tipo('d',T11)),`
	`T12 = array(56,T8),`
	`T13 = array(34,T12),`
	`asserta(tipo('c',T13)),`

	`.` `# fin del programa`

Example using SLY

```
ent -> linea ent
      | epsilon

linea -> tipo {lista.h := tipo.s;} lista ';'

tipo -> INT {tipo.s := contador++; escribe("T", tipo.s, " = entero");}
      | CHAR {tipo.s := contador++; escribe("T", tipo.s, " = caracter");}
      | FLOAT {tipo.s := contador++; escribe("T", tipo.s, " = flotante");}

lista -> {elm.h := lista.h;} elm {listaPrima.h := lista.h;} listaPrima

listaPrima -> ',' {elm.h := listaPrima.h;} elm {listaPrima1.h := listaPrima.h;} listaPrima1
           | epsilon

elm -> '*' {elm1.h := contador++; escribe("T", elm1.h, "=puntero(T", elm.h,")");} elm1
       | ID {dim.h := elm.h;} dim {escribe("asserta(tipo('", ID.lexval, "',T", dim.s,
")),");}

dim -> '[' NUM ']' {dim1.h := dim.h;} dim1 {dim.s := contador++; escribe("T", dim.s,
"=array(", NUM.lexval, ",T", dim1.s,"),");}
        | epsilon {dim.s := dim.h;}
```

Example using SLY

```
class CalcLexer(Lexer):
    tokens = {ID, CHAR, INT, FLOAT, NUM}
    ignore = ' \t'
    literals = { '*', ',', ';', '[',']'}

    @_(r'\d+')
        def NUM(self, t):
            t.value = int(t.value)
            return t
# Regular expression rules for tokens
    ID              = r'[a-zA-Z_][a-zA-Z0-9_]*'
    ID['char']      = CHAR
    ID['int']       = INT
    ID['float']     = FLOAT

    @_(r'\n+')
        def newline(self, t):
            self.lineno += t.value.count('\n')
        def error(self, t):
            print("Illegal character '%s'" % t.value[0])
            self.index += 1
```

Example using SLY

```
contador = -1

class CalcParser(Parser):
    tokens = CalcLexer.tokens

    def __init__(self):
        self.names = { }

    @_('linea ent')
        def ent(self, p):
            pass

    @_(' ')
        def ent(self, p):
            pass

     @_('tipo lista ";"')
        def linea(self, p):
        return p.tipo
```

```
    @_('INT')
        def tipo(self, p):
            global contador
            contador += 1
            print("T", contador, " = entero,")
            return contador
    @_('FLOAT')
        def tipo(self, p):
            global contador
            contador += 1
            print("T", contador, " = flotante,")
            return contador
    @_('CHAR')
        def tipo(self, p):
            global contador
            contador += 1
            print("T", contador, " = caracter,")
            return contador
```

Example using SLY

```
@_('empty5 elm empty4 listaPrima')
    def lista(self, p):
        return p[-2]
@_('')
    def empty4(self,p):
        return p[-3]

@_('')
    def empty5(self,p):
        return p[-1]

@_('"," empty3 elm empty2
    def listaPrima(self, p
        return p[-4]

@_('')
     def empty2(self,p):
        return p[-4]
```

```
@_('')
    def empty3(self,p):
        return p[-2]
@_('')
    def listaPrima(self, p):
        return p[-1]
@_('"*" empty1 elm')
    def elm(self, p):
    return p.empty1

@_('')
    def empty1(self,p):
    global contador
    contador += 1
    print("T", contador, " = puntero(T", p[-2], "),")
    return contador
```

Example using SLY

```
@_('ID empty0 dim')
    def elm(self, p):
        print("asserta(tipo('", p.ID, "', T", p.dim, ")),")

@_('')
    def empty0(self,p):
        return p[-2]

@_('"[" NUM "]" empty_1 dim')
    def dim(self, p):
        global contador
        contador += 1
        print("T", contador, "= array(", p.NUM, ",T", p.dim, "),")
        return contador

@_('')
    def empty_1(self,p):
        return p[-4]
```

Example using SLY

```
    @_('')    def dim(self,p):
      return p[-1]

if __name__ == '__main__':
   lexer = CalcLexer()
   parser = CalcParser()
   while True:
      try:
         text = input('data type list > ')
      except EOFError:
         break
      if text:
         parser.parse(lexer.tokenize(text))
```

Scheme implementation in a recursive descent translator

- Inherited attributes are the input parameters of the function
- The synthesized attributes are the output parameters of the function.
- The symbols attributes of the body of a rule is stored as variables local to the function.
- We must save the lexical values before squaring the token.

Adapt schemes descending

- Removing left recursion

$$A \rightarrow A\,\alpha \mid \beta \quad \Rightarrow \quad A \rightarrow \beta\ R \qquad R \rightarrow \alpha\ R \mid \xi$$

- Factorizing

$$A \rightarrow \alpha\,\beta_1 \mid \alpha\,\beta_2 \mid \ldots \mid \alpha\,\beta_n \mid \Gamma \quad \Rightarrow \quad A \rightarrow \alpha\ A' \mid \Gamma \qquad A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

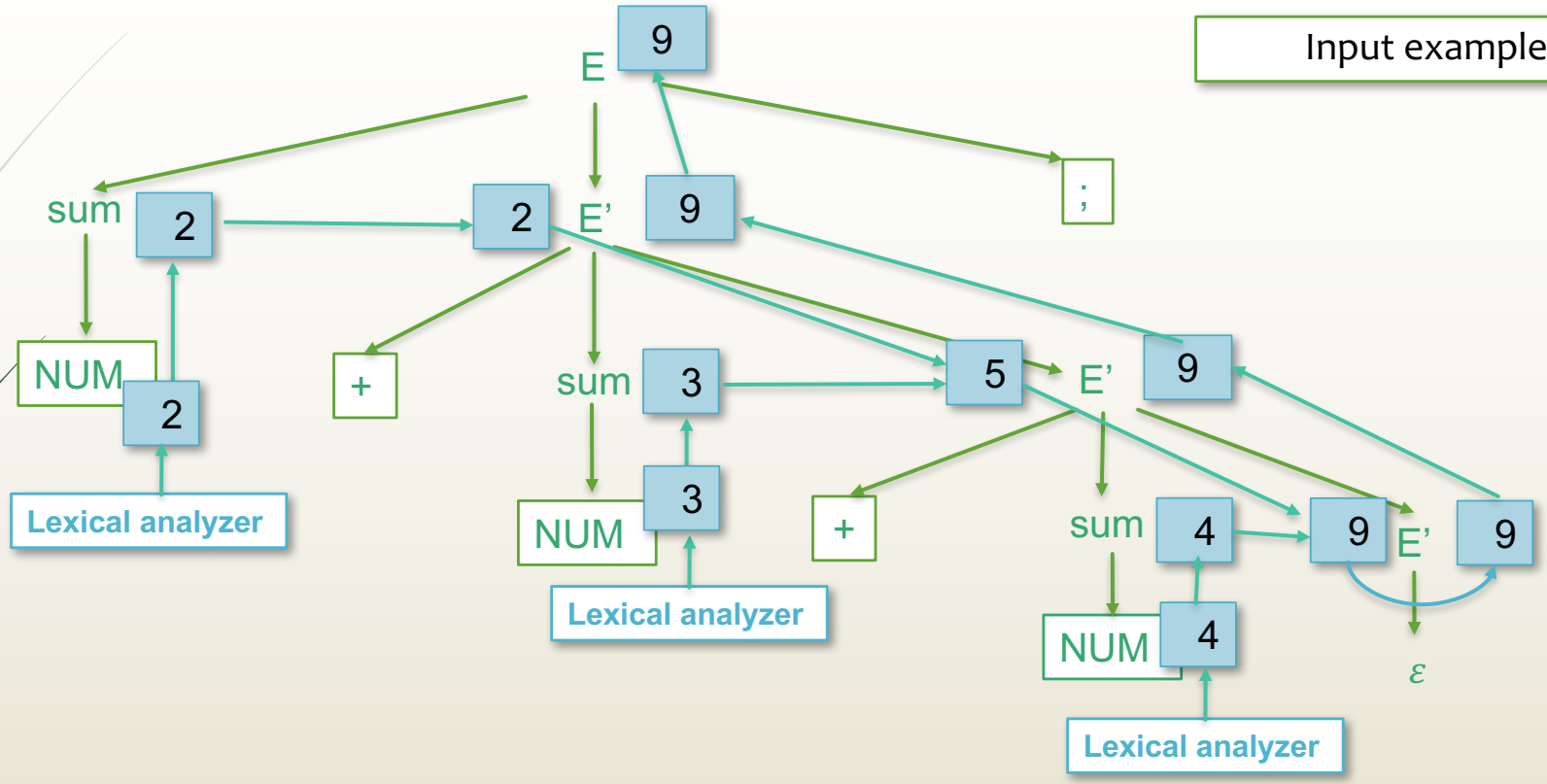
Adapt schemes descending

- Although the original scheme only have synthesized attributes, to adapt to the downward, there may be inherited attributes.
- Example: calculator

```
E -> E1 '+' Sum {E.s := E1.s + Sum.s;}
E -> E1 '-' Sum {E.s := E1.s - Sum.s;}
E -> Sum        {E.s := Sum.s;}
Sum -> NUM      {Sum.s := NUM.lexval;}
```

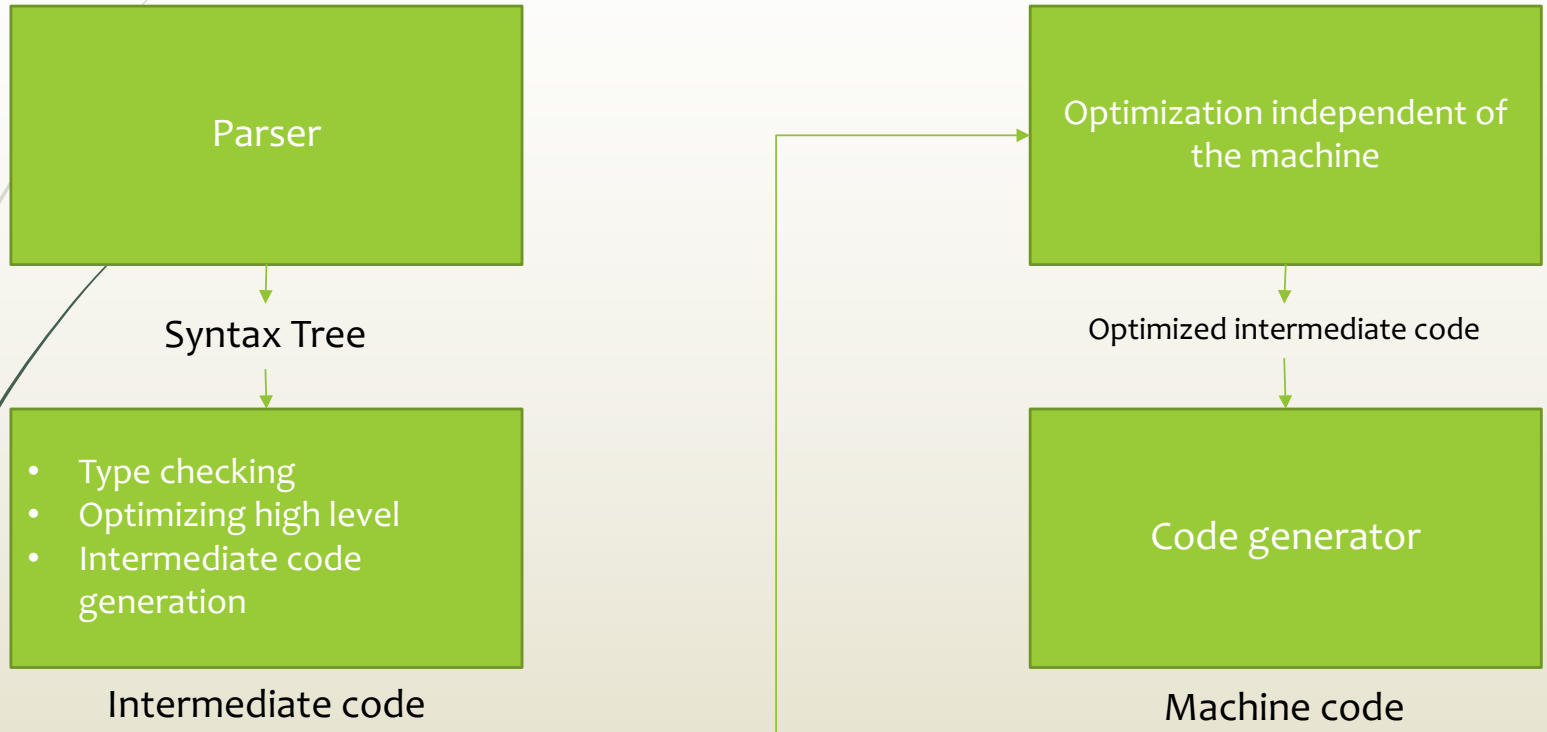
```
E -> Sum {E'.h := Sum.s;} E' {E.s := E'.s;}
E' ->'+' Sum {E'1.h := E'.h + Sum.s;} E'1 {E.s := E'1.s;}
E' ->'-' Sum {E'1.h := E'.h - Sum.s;} E'1 {E.s := E'1.s;}
E' -> épsilon {E'.s := E'.h;}
Sum -> NUM    {Sum.s := NUM.lexval;}
```

Decorated tree example



Abstract Syntax Tree (AST)

Intermediate code generation

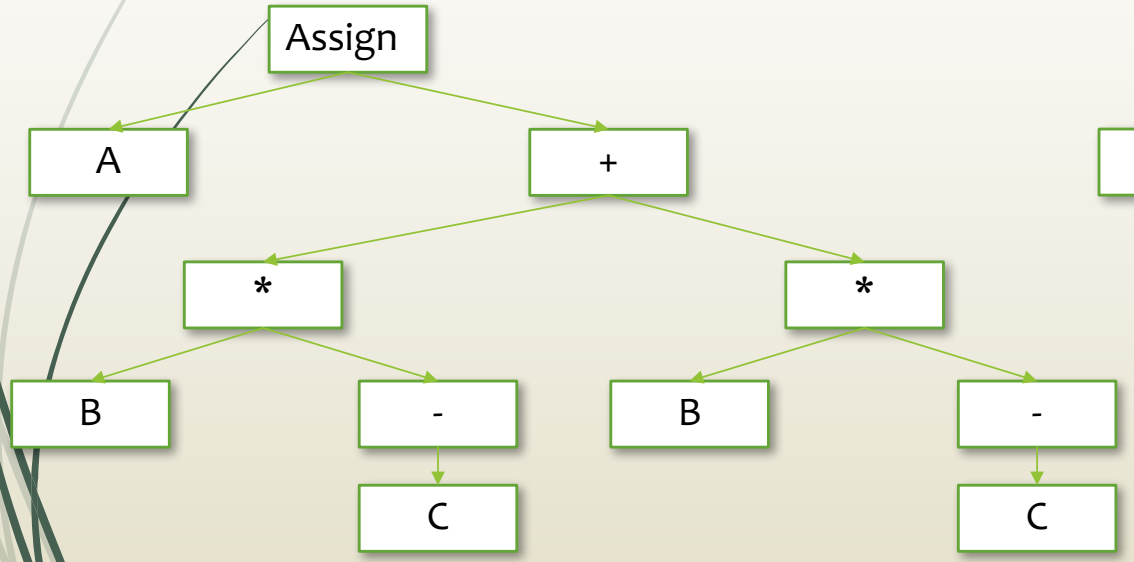


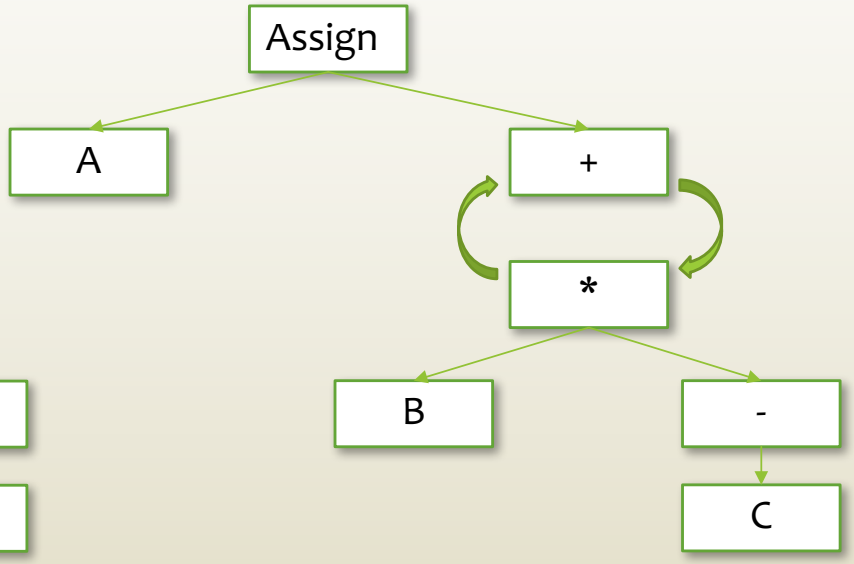
First of intermediate representation (IR)

- Syntactic Trees (ST), also called **Abstract Syntax Tree (AST)** and its close relative: Acyclic directed graphs (ADG):
  - They are the 1st intermediate representations
  - Suitable to represent arithmetic and logical expressions, control statements and declarations
  - The type checking is usually done on this representation.

Syntax Trees

- Syntax Tree (ST) and Acyclic Directed Graphs (ADG) for de input:
  - **A = B * - C+ B * - C**





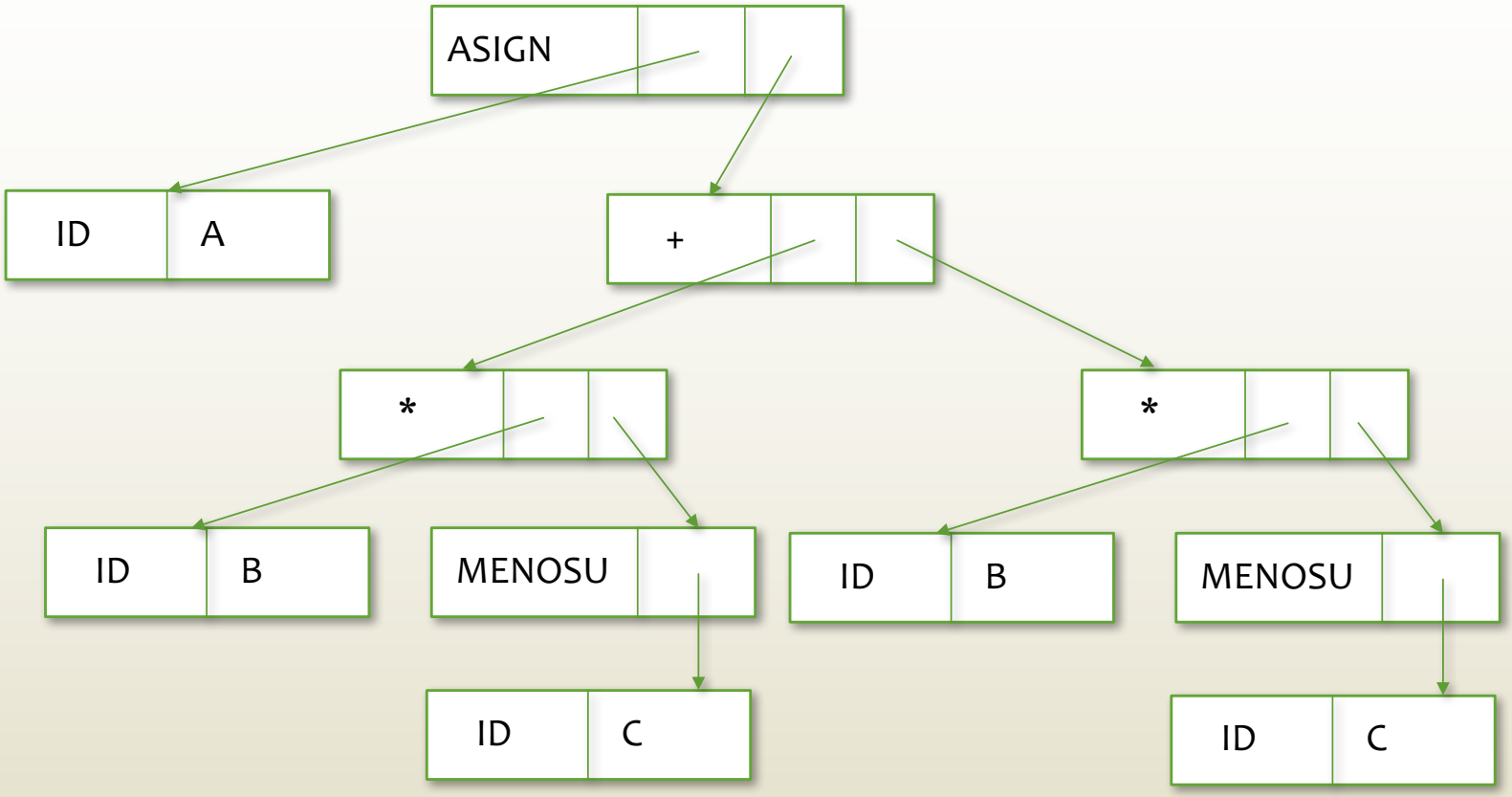
Translation scheme for ST construction

- Translation schemes for assignment statements:
  - If the ***HazNodo*** functions always create a new entry, the result is a syntax tree
  - If ***HazNodo*** functions check that entries are already created, in order to avoid duplications, the result will be an acyclic graph

Translation scheme to generate a ST or ADG using assignment statements

Grammar	Semantic rules
S→ID ':=' E	`S.ptr := hazNodoAsign(hazNodoId(ID.lexval), E.ptr)`
E→E1 + E2	`E.ptr := hazNodoSuma(E1.ptr, E2.ptr)`
E→E1 – E2	`E.Ptr := hazNodoResta(E1.ptr, E2.ptr)`
E→E1 * E2	`E.ptr := hazNodoProd(E1.ptr, E2.ptr)`
E→- E1	`E.ptr := hazNodoMenosU(E1.ptr)`
E→(E1)	`E.ptr := E1.ptr`
E→ID	`E.ptr := hazNodoId(ID.lexval)`
E→NUM	`E.ptr := hazNodoNum(Num.lexval)`

Resulting ST



Example: creation of ST

- Translation scheme

```
ent -> E {escribe(E.s)} ent
     | ε
E -> T {Re.h := T.s} Re {E.s := Re.s}
Re -> + T {Re1 := haznodo(+, Re.h, T.s)} Re1 {Re.s := Re1.s}
     | - T {Re1 := haznodo(-, Re.h, T.s)} Re1 {Re.s := Re1.s}
     | ε {Re.s := Re.h}
T -> P {Rt.h := P.s} Rt {T.s := Rt.s}
Rt -> * P {Rt1.h := haznodo(*, Rt.h, P.s)} Rt1 {Rt.s := Rt1.s}
     | / P {Rt1.h := haznodo(/, Rt.h, P.s)} Rt1 {Rt.s := Rt1.s}
     | ε {Rt.s := Rt.h}
P -> F {Rp.h := F.s} Rp {P.s := Rp.s}
Rp -> ↑ F {Rp1.h:= F.s} Rp1 {Rp.s := haznodo(↑, Rp.h, Rp1.s)}
     | ε {Rp.s := Rp.h}
F -> NUM {F.s := hazhoja(NUM, NUM.lexval)}
     | (E) {F.s := E.s}
     | - F1 {F.s := haznodo(~, F1.s)}
```

Example: creation of ST

Global variables and extra functions

```
from sly import Lexer

Max = 80
line = ""
ta = None
ind = 0
tokens = None
tokenlist = None

def esp(level, old, ne):
    global line
    if level > 1:
        if old:
            line = line[0:(level - 2 )] + "|   "
        else:
            line = line[0:(level - 2)] + "    "
    if ne:
        line = line[0:level] + "|-->"
    else:
        line = line[0:level] + "|L -->"
    return line
```

```
def cuadra(t):
    global ta, ind, tokenlist
    if ta.type == t:
        ind += 1
        if ind < len(tokenlist):
            ta = tokenlist[ind]
    else:
        print("No cuadra token = ", ta.value)
```

Example: creation of ST

Classes

```
class Node():
    def escribe(level, old, ne):
        pass

class InternalNode(Node):
    def __init__(self, e, p1, p2):
        self.label = e
        self.pn1 = p1
        self.pn2 = p2
    def escribe(self, level, old, ne):
        print(esp(level, old, ne), "node (", self.label, ")")
        self.pn1.escribe(level + 2, ne, True);
        self.pn2.escribe(level + 2, ne, False);

class UniqueNode(Node):
    def __init__(self, e, p1):
        self.label = e;
        self.pn1 = p1;
    def escribe(self, level, old, ne):
        print(esp(level, old, ne), "node (", self.label, ")")
        self.pn1.escribe(level + 2, ne, True);
```

Example: creation of ST

Classes

```
class NodeId(Node):
    def __init__(self, n):
        self.name = n
    def escribe(self, level, old, ne):
        print(esp(level, old, ne), "ID(", self.name, ")")

class NodeNum(Node):
    def __init__(self, v):
        self.value = v
    def escribe(self, level, old, ne):
        print(esp(level, old, ne), "NUM (", self.value, ")")
```

Example: creation of ST

Lexical analyzer

```
class CalcLexer(Lexer):
    tokens = {ID, ASIG, NUMBER}
    ignore = ' \t'
    literals = {';', '+', '-', '*', '/', '^', '(', ')'}

    # Regular expression rules for tokens
    ID   = r'[a-zA-Z_][a-zA-Z0-9_]*'
    ASIG = r':='        @_(r'\d+')

    def NUMBER(self, t):
        t.value = int(t.value)
        return t

    @_(r'\n+')
        def newline(self, t):
        self.lineno += t.value.count('\n')
    def error(self, t):
        print("Illegal character '%s'" % t.value[0])
        self.index += 1
```

Example: creation of ST

Parser (top - down)

```
def F():
     global ta, tokens
     if ta.type == tokens[2]: #NUMBER
          f_s = NodeNum(ta.value)
          cuadra("NUMBER")
          return f_s
     else:
          if ta.type == tokens[1]: #ID
               f_s = NodeId(ta.value)
               cuadra("ID")
               return f_s
          else:
               if ta.type == '(':
                    cuadra("(")
                    f_s = E()
                    cuadra(")")
                    return f_s
               else:
                    if ta.type == '-':
                         cuadra('-')
                         f1_s = F();
                         f_s = UniqueNode('~', f1_s)
                         return f_s
                    else:
                         print("Error en F()")
```

Example: creation of ST

Parser (top - down)

```
def Rp(rp_h):
    global ta
    if ta.type == '^':
        cuadra("^")
        f_s = F()
        rp1_s = Rp(f_s)
        rp_s = InternalNode("^", rp_h, rp1_s)
        return rp_s
    else:
        return rp_h
#end of Rp

def P():
    global ta, tokens
    if ta.type == tokens[2] or ta.type == tokens[1] or ta.type == '-' or ta.type == '(':
        f_s = F()
        p_s = Rp(f_s)
        return p_s
    else:
        print("Error en P()");
#end of P
```

Example: creation of ST

Parser (top - down)

```
def Rt(rt_h):
     global ta
     if ta.type == '*':
          cuadra("*")
          p_s = P()
          rt1_h = InternalNode("*", rt_h, p_s)
          rt_s = Rt(rt1_h)
          return rt_s
     else:
          if ta.type == '/':
               cuadra('/')
               p_s = P()
               rt1_h = InternalNode('/', rt_h, p_s)
               rt_s = Rt(rt1_h)
               return rt_s
          else:
               return rt_h;
#end of Rt
def T():
     global ta,tokens
     if ta.type == tokens[2] or ta.type == tokens[1] or ta.type == '-' or ta.type == '(':
          p_s = P()
          t_s = Rt(p_s)
          return t_s
     else:
          print("Error en T()")
#end of T
```

Example: creation of ST

Parser (top - down)

```
def Re(re_h):
     global ta
     if ta.type == '+':
          cuadra("+")
          t_s = T()
          re1_h = InternalNode('+', re_h, t_s)
          re_s = Re(re1_h)
          return re_s
     else:
          if ta.type == '-':
               cuadra("-")
               t_s = T()
               re1_h = InternalNode('-', re_h, t_s)
               re_s = Re(re1_h)
               return re_s
          else:
               return re_h;
#end of Re
def E():
     global ta, tokens
     if ta.type == tokens[2] or ta.type == tokens[1] or ta.type == '-' or ta.type == '(':
          t_s = T()
          e_s = Re(t_s)
          return e_s
     else:
          print("Error en E()")
#end of E
```

Example: creation of ST

Parser (top - down)

```
def entrada():
    global ta, ind, tokenlist, tokens
    if ta.type == tokens[1]:
        na = ta.value
        try:
            cuadra("ID")
            cuadra("ASIG")
            es = E()
            es1 = InternalNode('=', NodeId(na), es)
            es1.escribe(0,False, False)
        except EOFError:
            print("Error en entrada(). Salto al ;")
            while ta.type != ';' and ind < len(tokenlist):
                ind += 1
                ta = tokenlist[ind]
            cuadra(';')
            entrada()
    else:
        if ind < len(tokenlist):
            print("Error en entrada()")
#end of entrada
```

Example: creation of ST

Parser (top - down)

```
if __name__ == '__main__':
    global ta, tokens, line
    lexer = CalcLexer()
    tokens = CalcLexer.tokens
    tokens = sorted(tokens)
    print("Comienza el programa\n")
    print("Teclee expresiones aritmeticas terminadas en punto y coma\n")
    while True:
        try:
            data = input('data type list > ')
        except EOFError:
            break
        if data:
            tokenlist = list(lexer.tokenize(data))
            ta = tokenlist[ind]
            entrada()
        ind = 0
        line = ""
```