Item 1: Syntax-directed translation

Mari Paz Guerrero Lebrero

Grado en Ingeniería Informática

Curso 2017/2018

Semantic

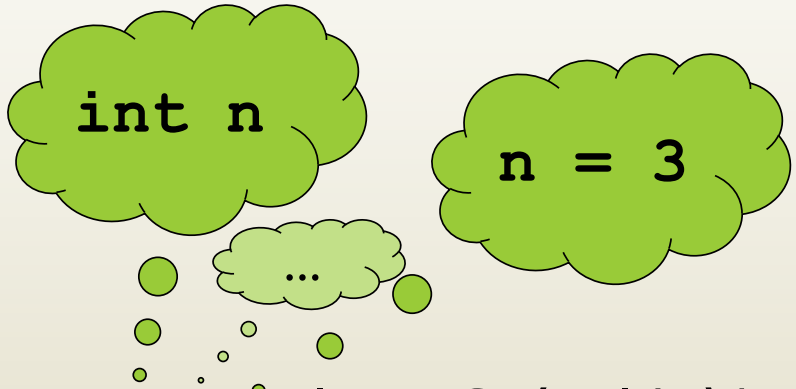
- The compiler should keep invariant meaning of the program, regardless of the language syntax in which it is written
- The semantics informally is described in natural language for most programming languages ; two compilers might give different results for the same program source
- Unlike what happens 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

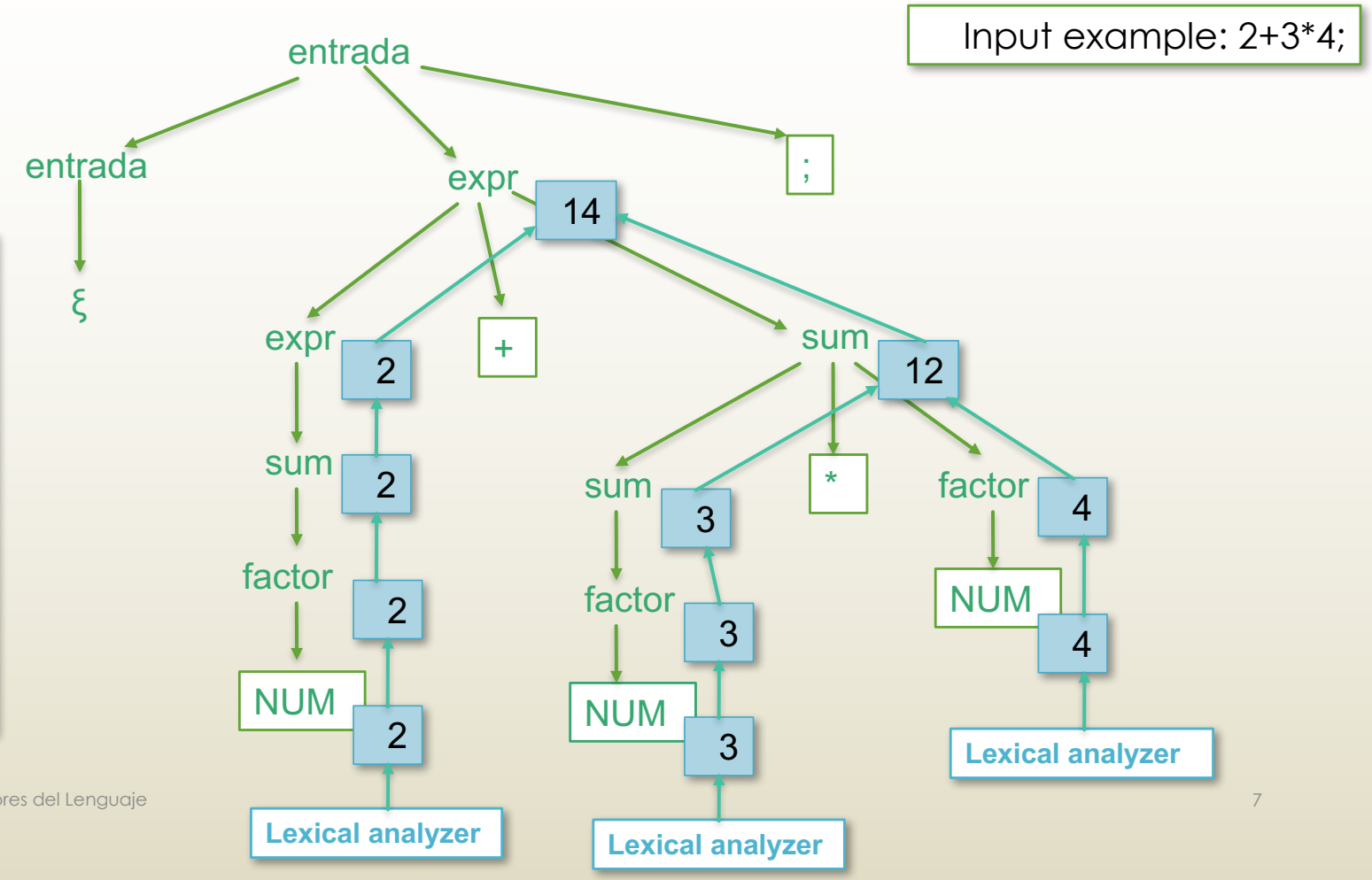
To all this we call a syntax-directed definition

Decorated tree example

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 → $expr_1$ + 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 (Bison)

```
%union {
  int s;
}

%token NUM

%type <s> NUM expr sum factor

%%
entrada : entrada expr ';' {cout << $2 << endl;}
        |
;
expr : expr '+' sum    {$$ = $1 + $3;}
     | sum             {$$ = $1;}
;
sum : sum '*' factor   {$$ = $1 * $3;}
    | factor           {$$ = $1;}
;
factor : '-' factor    {$$ = - $2;}
       | NUM           {$$ = $1;}
       | '(' expr ')'  {$$ = $2;}
;
```

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 Bison

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 **synthesize**d 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 (Bison)

```
%{
#include <iostream>
#include <cmath>
using namespace std;

int yyerror(const char * msj);
int yylex(void);

#define YYERROR_VERBOSE
%}

%union {
float vf;
struct {
    int v;
    int lon;
} viyl;
} int vi;

%type <vf> num
%type <viyl> sec
%type <vi> dig
```

```
%%
entrada : num {cout << "El valor = " << $1 << endl;}
;

num : sec '.' sec {$$ = $1.v + float($3.v)/
                                pow(2.0,$3.lon);}
;

sec : secdig {$$.v=$1.v*2+$2;
              $$.lon = $1.lon + 1;}
    | dig {$$.v = $1;
           $$.lon = 1;}
;

dig:'0' {$$ = 0;}
   |'1' {$$ = 1;}
;
```

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

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 (Bison)

```
%{
#include <iostream>
#include <string>
#include <map>

using namespace std;

#include "nodo.h"

map <string,Nodo*> tabla;
int yyerror(const char * msj);
int yylex(void);

#define YYERROR_VERBOSE
%}

%union {
string * nom;
} Nodo * tip;

%token ID INTEGER FLOAT CHAR
%type <nom> ID
%type <tip> tipo
```

```
%%
entrada : definicion entrada
    |
;
definicion : tipo {$<tip>$ = $1;} lista ';'
;
tipo : INTEGER {$$ = new NodoEntero;}
     | FLOAT {$$ = new NodoFlotante;}
     | CHAR {$$ = new NodoChar;}
;
lista : {$<tip>$ = $<tip>0;} elm {$<tip>$ = $<tip>0;} resto ;
resto :',' {$<tip>$ = $<tip>0;} elm {$<tip>$ = $<tip>0;} resto
      |
;
elm : '*' {$<tip>$ = new NodoPuntero($<tip>0);} elm
    | ID {tabla[*$1] = $<tip>0;}
;
```

Ascending implementation (Bison)

```
%%
int main(){
  cout << "Programa que lee definiciones de variables\n";
  yyparse();
  cout << "Final de la entrada estas son las variables\n";
  for (map<string,Nodo*>::iterator i = tabla.begin(); i !=
       tabla.end(); i++)
  {
    cout << i->first << " es un " << i->second->escribir() <<
         endl;
  } // fin del for
  cout << "Final del programa\n";
} // fin del main()

int yyerror(const char * msj) {
  cerr << msj << endl;
  return 1;
} // fin de yyerror()
```

Ascending implementation (FLEX)

```
%{
#include <iostream>
#include <string>

using namespace std;

#include "nodo.h"
#include "parser.tab.h"
%}

%%
[ \n\t]+     ;
[;,*]        return *yytext;
char         return CHAR;
int          return INT;
float        return FLOAT;
[a-zA-Z_]+   {yylval.nom = new string(yytext); return ID;}
.            {cerr << "Error lexico, caracter (" << yytext << ") raro\n";}
```

Ascending implementation

```
struct Nodo {
virtual string escribir(void) = 0;
};

struct NodoEntero: public Nodo {
  public:
    string escribir(void) {return string("Entero");}
};

struct NodoFlotante: public Nodo {
  string escribir(void) {return string("Flotante");}
};

struct NodoChar: public Nodo {
  string escribir(void) {return string("Caracter");}
};

struct NodoPuntero: public Nodo {
  Nodo * pn;
  NodoPuntero(Nodo * puntNodo) : pn(puntNodo) {}
  string escribir(void) {return string("Puntero a ")+pn->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) $\subset \xi$, 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

```
#include <iostream>
#include <string>
#include <map>
 using namespace std;
#include "cabecera.h"
#include "nodo.h"

map <string,Nodo*> tabla;
void cuadra(int obj);
void yyerror(const string & msj);
int yylex(void);
extern char * yytext;

void entrada(void);
void definicion(void);
Nodo* tipo(void);
void lista(Nodo * listah);
void resto(Nodo * restoh);
void elm(Nodo* elmh);

int ta;
string * yylval;
```

- 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

```
void entrada(void) {
  if (ta == INTEGER || ta == FLOAT || ta == CHAR) {
    definicion();
    entrada();
  }
  else if (ta == 0) // 0 es el fin de archivo para flex
    return;         // entrada -> epsilon
  else
    yyerror("en entrada");
} // fin de entrada()

void definicion(void) {
  if (ta == INTEGER || ta == FLOAT || ta == CHAR) {
    Nodo * tipos = tipo();
    Nodo * listah = tipos;
    lista(listah);
    cuadra(';');
  }
  else
    yyerror("en definicion");
} // fin de definicion()
```

```
Entrada → definicion entrada
        | ξ

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

Down recursive implementation

```
Nodo * tipo(void) {
  if (ta == INTEGER) {
    cuadra(INTEGER);
    return new NodoEntero;
  }
  else if (ta == FLOAT) {
    cuadra(FLOAT);
    return new NodoFlotante;
  }
  else if (ta == CHAR) {
    cuadra(CHAR);
    return new NodoChar;
  }
  else yyerror("en tipo");
} // fin de tipo()

void lista(Nodo * listah) {
  if (ta == '*' || ta == ID) {
    elm(listah);
    resto(listah);
  }
  else yyerror("en lista");
} // fin de 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
```

Down recursive implementation

```
void resto(Nodo *restoh){
  if(ta == ',' ){
    cuadra(',');
    elm(restoh);
    resto(restoh);
  }
  else if(ta == ';')
    return;
  else
    yyerror("en resto");
} //fin de resto()

void elm(Nodo *elmh){
  if(ta == '*'){
    cuadra('*');
    Nodo * elm1h = new NodoPuntero(elmh);
    elm(elm1h);
  }
  else if (ta == ID){
    string *IDlexval = yyleval;
    cuadra(ID);
    tabla[*IDlexval] = elmh;
  }
  else yyerror("en elm");
}
```

```
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

```
int main() {
  cout << "Programa que lee definiciones de variables\n";
  ta = yylex();
  entrada();
  cout << "Final de la entrada estas son las variables\n";
  for (map<string,Nodo*>::iterator i = tabla.begin(); i != tabla.end(); i++) {
    cout << i->first << " es un " << i->second->escribir() << endl;
  } // fin del for
  cout << "Final del programa\n";
}//fin del main()

void yyerror(const string & msj) {
  cerr << "error sintactico " << msj << endl;
  exit(EXIT_FAILURE);
} // fin de yyerror()

void cuadra(int obj) {
  if (ta == obj){
    cout << "cuadro ta=" << ta << " (" << yytext << ")";
    ta = yylex();
    cout << " ==> nuevo ta=" << ta << " (" << yytext << ")\n";
  }
  else yyerror("en cuadra");
} // fin de cuadra()
```

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)

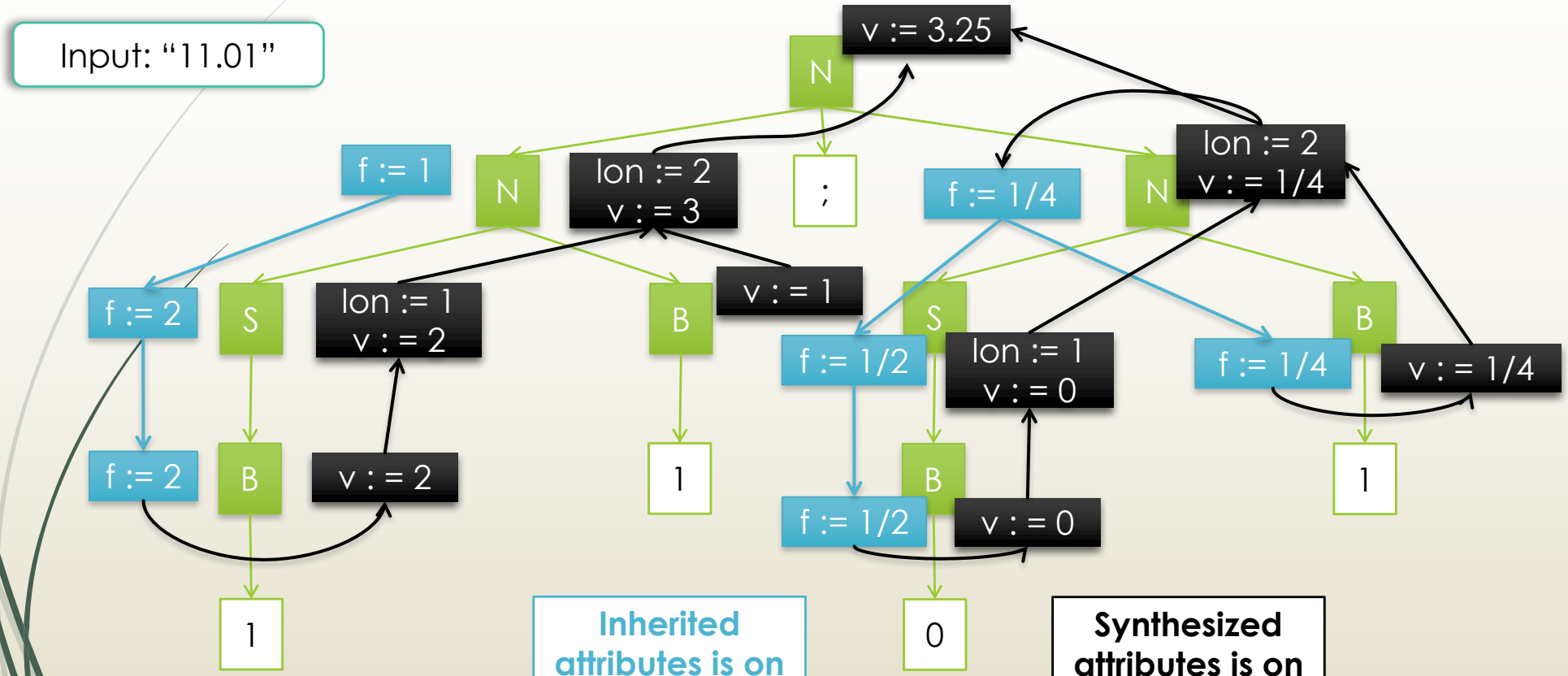
Definition that is not L-Attribute

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

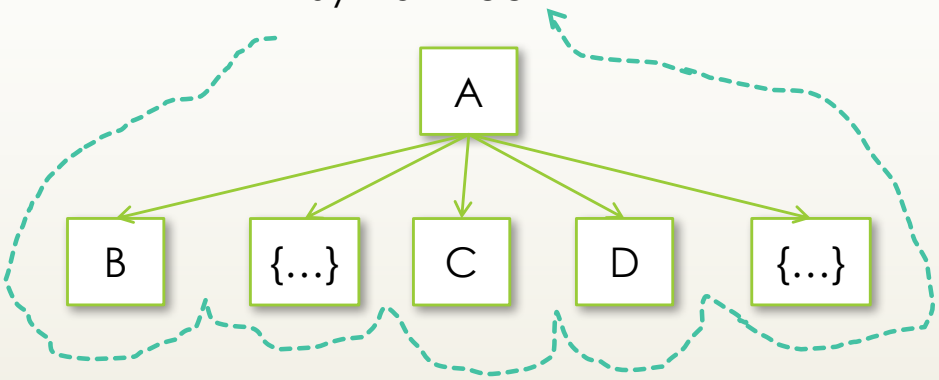
- Measures using synthesized attributes must be placed somewhere after the symbol
  - To remember this limitation: the descending implementation, the synthesized attributes of certain symbol are the result of the function corresponding to the symbol; so they can only be used after calls to that function.
- 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 (Bison)

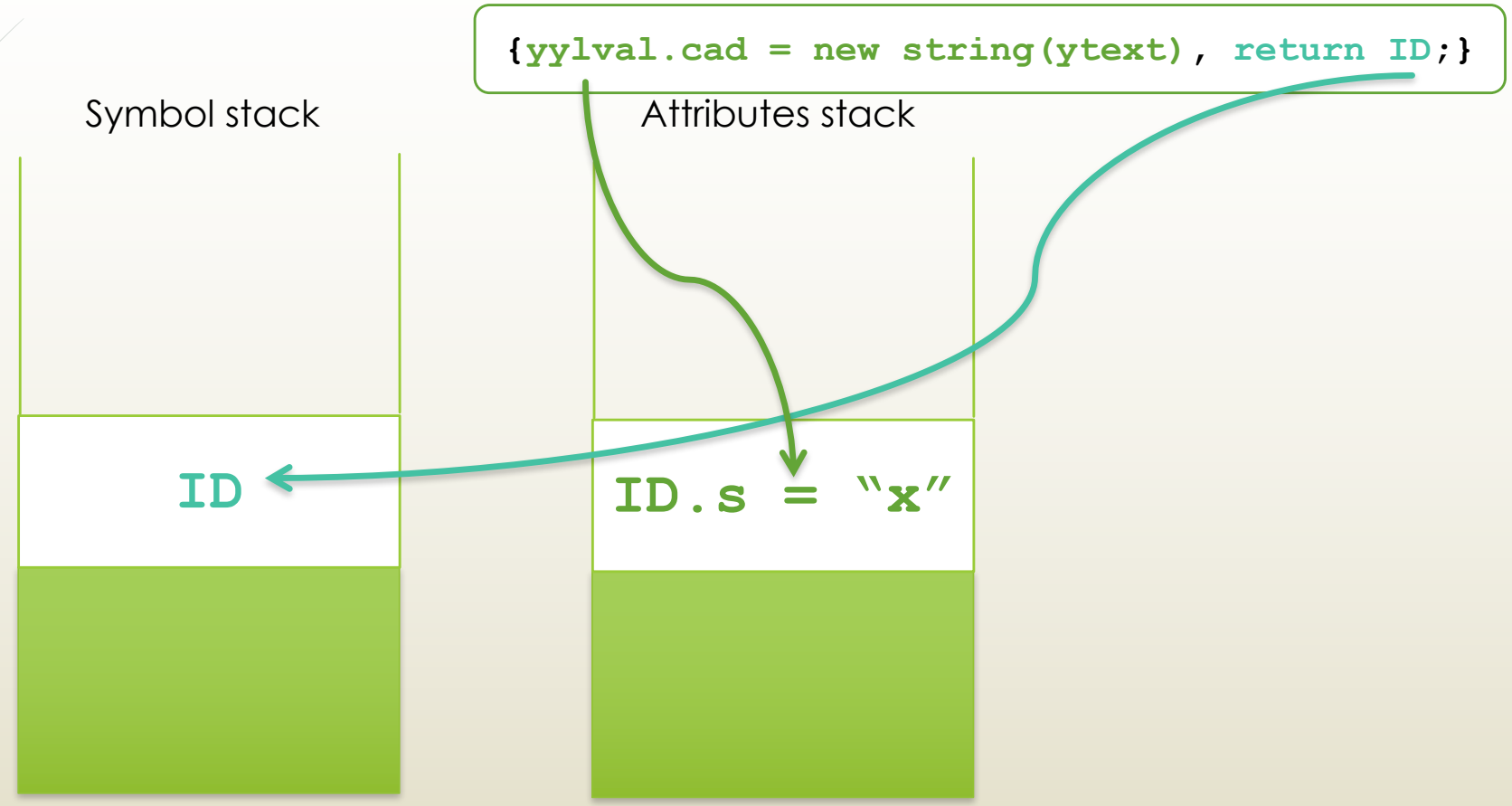
- Attributes stack is used in parallel with the symbols stack(actually, a stack of states).
- The type of each element of that attributes stack is a 'union {...}' as defined in Bison with '% union {...}' command
- In addition, Bison defines a variable called *yylval* with the same type of union {...} and links with the lexical analyser.
- Flex must place the lexical value (synthesized attribute tokens) in this variable and then return the token:

```
[a-z]+ {yylval.cad = new string(ytext), return ID;}
```

Actions during a shift

- The algorithm works by Bison 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.

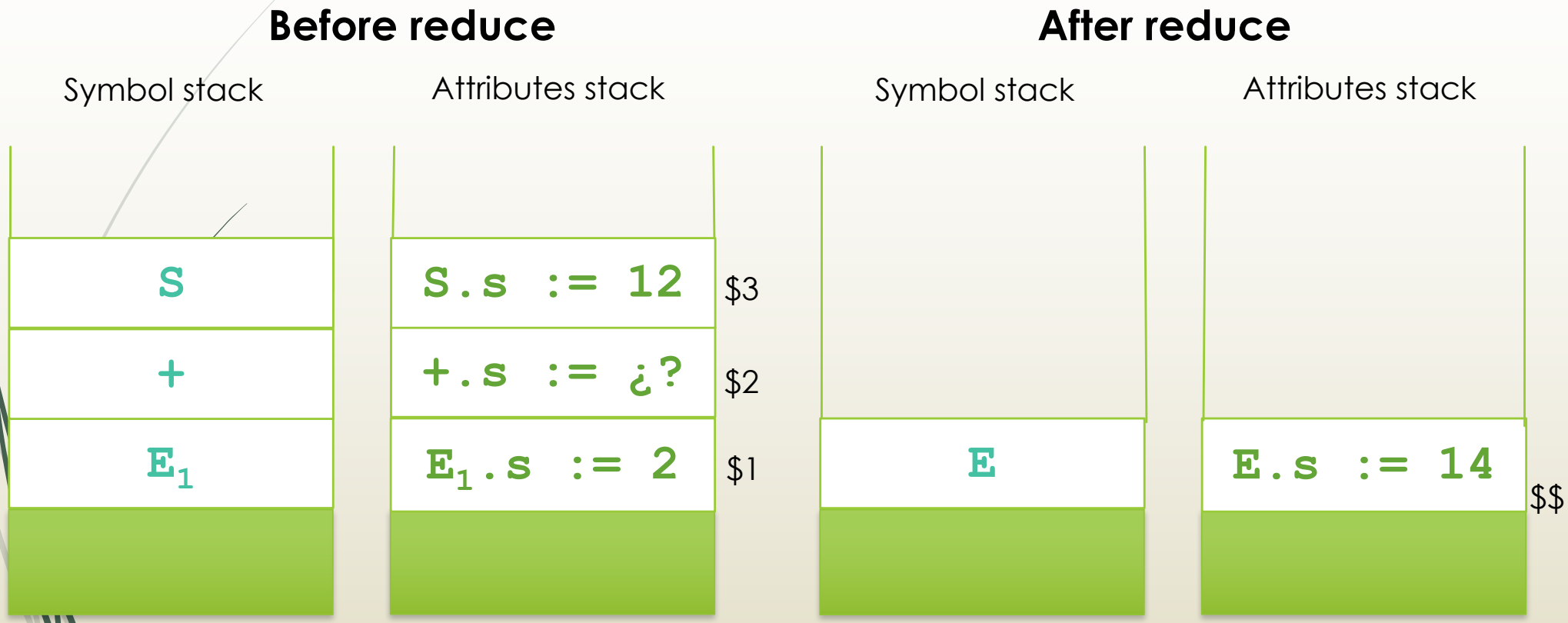
Actions during a shift



Actions during a reduce

- For a reduces by rule **A → a**, symbols of symbols stack (as many as a 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 Bison: E: E1 '+' S {$$ = $ 1 + $ 3;}
  - Semantic action {$$ = $1 + $3;} is triggered during reduced.
  - $$ indicates synthesized attribute of the header (would be placed on the attribute stack after reduction)
  - $1 and $3 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), Bison performs automatically a transformation inventing a new symbol and a new rule, so that just shooting with reduced.
- Example: A: X {printf ("hello\n");} Y Z
  - It automatically become the following: A → X @1 Y Z
  - @1 → ξ {printf ("hello\n");}
  - Being @1 a new symbol generated by Bison, and so that the semantic action ends at the end of a rule is triggered when the rule is reduced
- You can check with: bison -d -v parser.y; less parser.output

Example using Bison

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 Bison

```
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 Bison

```
%{
#include <iostream>
#include <string>
using namespace std;
#include "parser.tab.h"
%}

%%
"//".*      ;
[ \n\t]+    ;
[,*\[\];]   return *yytext;
int         return INT;
char        return CHAR;
float       return FLOAT;
[a-zA-Z]+   {yylval.cad=new string(yytext); return ID;}
[0-9]+      {yylval.cad=new string(yytext); return NUM;}
.           {cerr<<"Error léxico " << yytext << endl;}
```

Example using Bison

```
%{
#include <iostream>
#include <string>
using namespace std;
int yylex(void);
int yyerror(const char* s);
int contador=0;
%}

%union{
string* cad;
int val;
}
%token ID NUM CHAR FLOAT INT
%type <val> dim tipo
%type <cad> ID NUM
%%
```

Example using Bison

```
ent: linea ent
   |
;
linea: tipo {$<val>$ = $1;} lista ';'
;
tipo: INT {$$ = contador++; cout << "T" << $$ << " = entero,“ << endl;}
    | CHAR {$$ = contador++; cout << "T" << $$ << " = caracter,“ << endl;}
    | FLOAT {$$ = contador++; cout << "T" << $$ << " = flotante,“ << endl;}
;
lista: {$<val>$ = $<val>0;} elm {$<val>$ = $<val>0;} listaPrima ;
listaPrima: ',' {$<val>$ = $<val>0;} elm {$<val>$ = $<val>0;} listaPrima
          |
;
elm: '*' {$<val>$ = contador++;cout << "T" << $<val>$ << " = puntero (T" <<
$<val>0 << ")" << endl; } elm
   | ID {$<val>$ = $<val>0;} dim {cout << "asserta(tipo('" << *$1 << "',T"
<< $3 << ")),“ << endl;}
;
```

Example using Bison

```
dim: '[' NUM ']' {$<val>$ = $<val>0;} dim {$$ = contador++; cout << "T" <<
$$ << " = array(" << *$2 << ",T" << $5 << "),“ << endl; }
  |{$$ = $<val>0;}
;
%%
int main() {
cout << "# comienza el programa“ << endl;
yyparse();
cout << ".“ << endl;
cout << "#fin del programa“ << endl; return EXIT_SUCCESS;
}

int yyerror(const char* s) {
cout << s << endl;
}
```

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

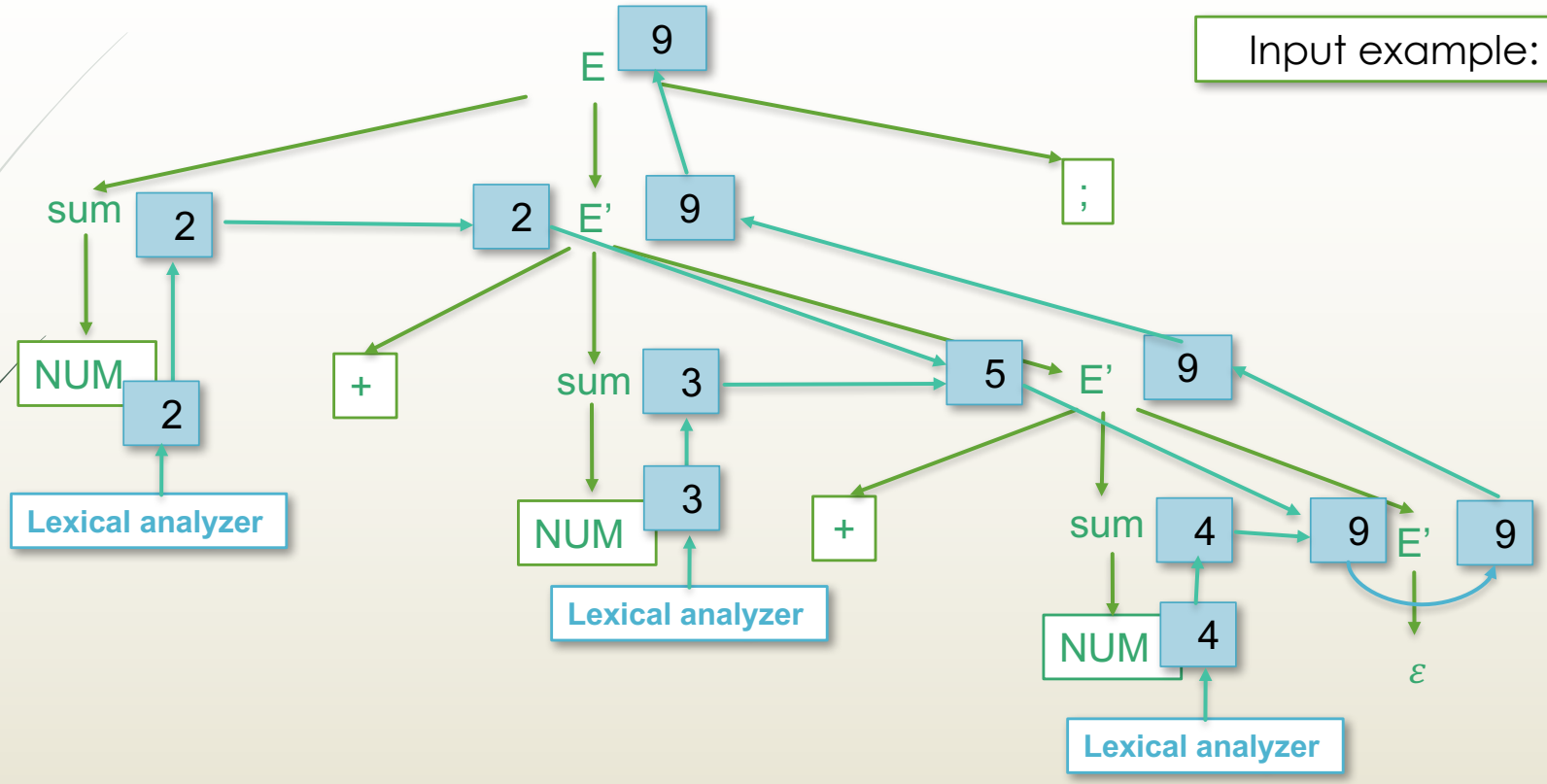
- Although the original scheme only have synthesized attributes, to adapt to the downward, there may be inherited attributes.
- Example: calculator

```
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;}
```

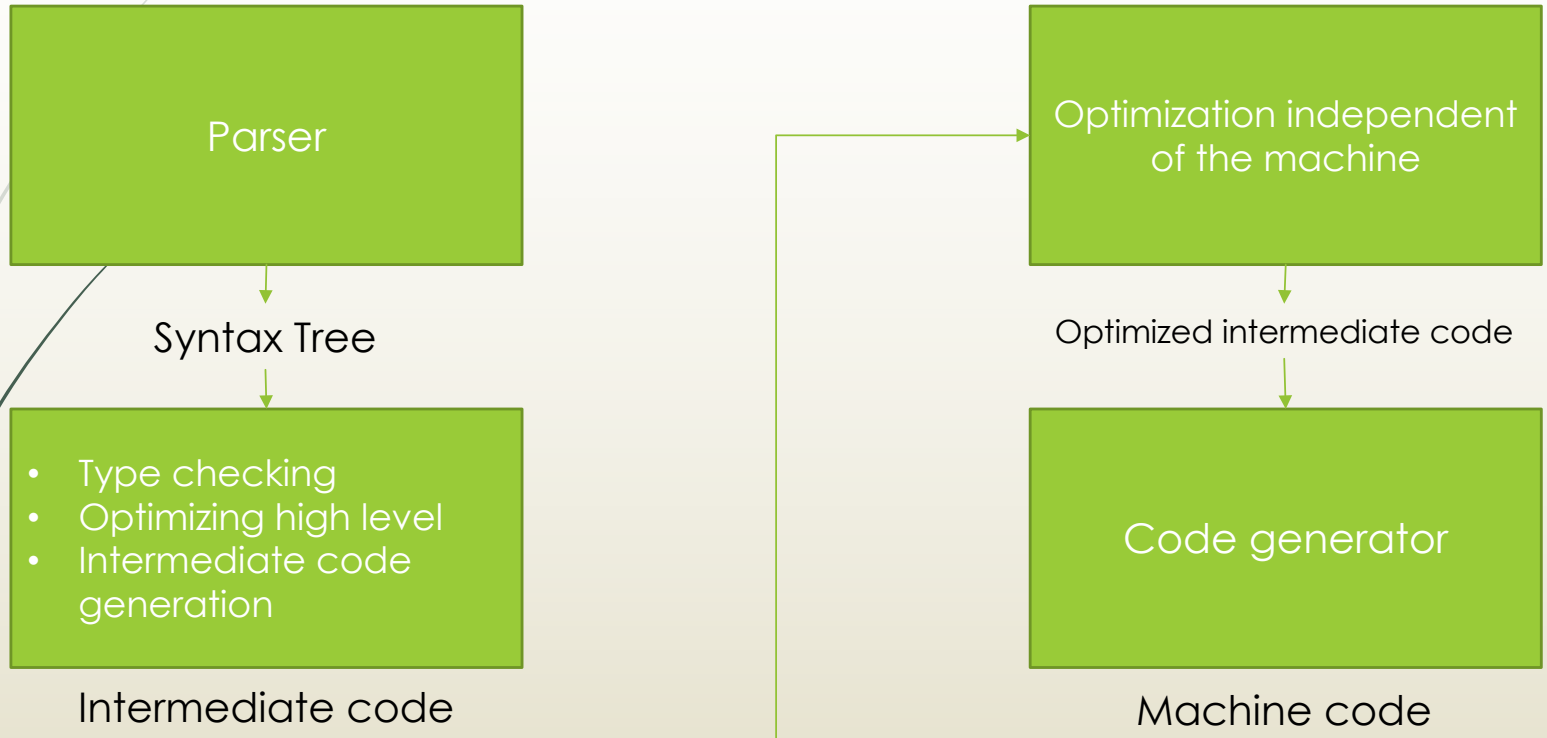
Decorated tree example

Input example: 2+3+4;



Abstract Syntax Tree (AST)

Intermediate code generation

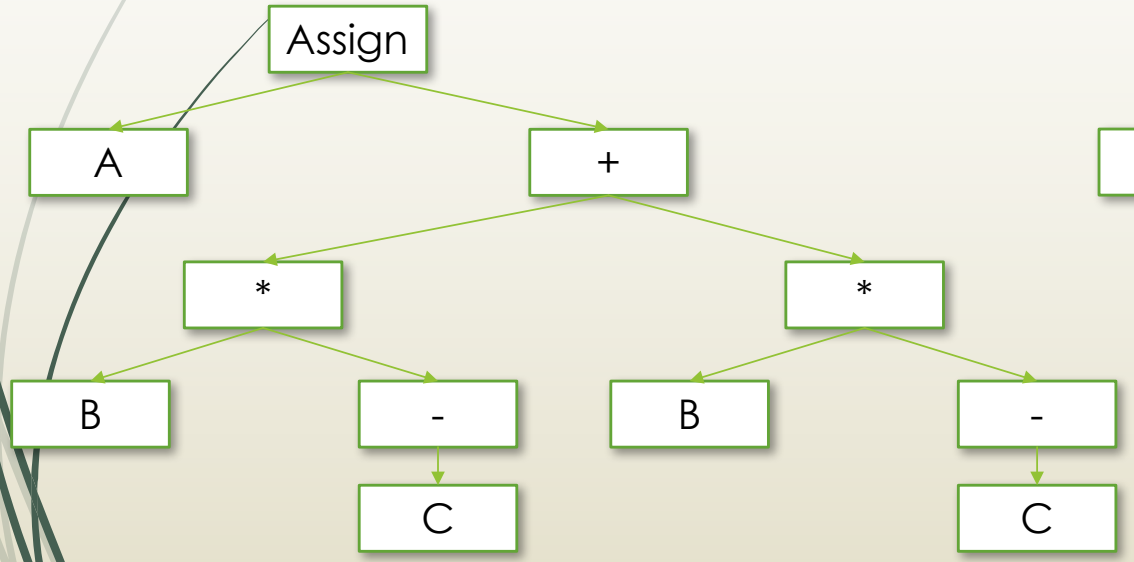


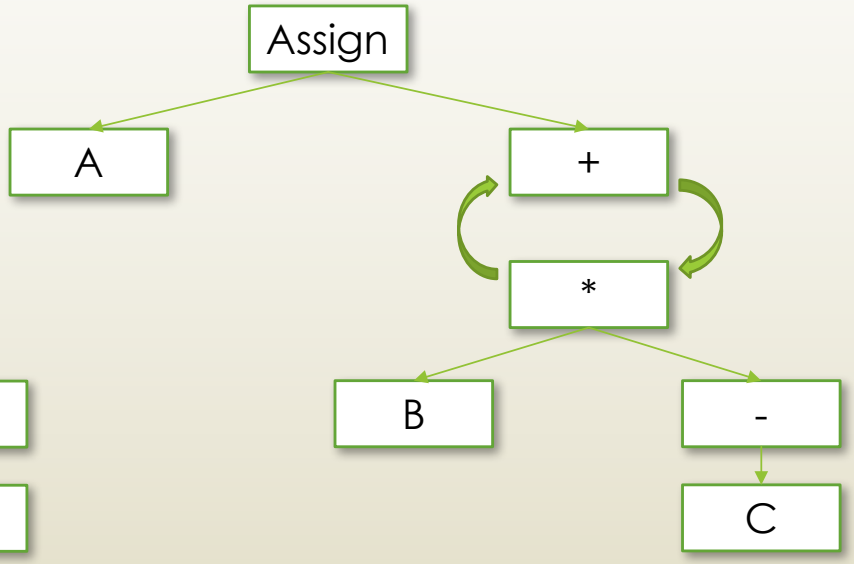
First of intermediate representation (IR)

- 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**





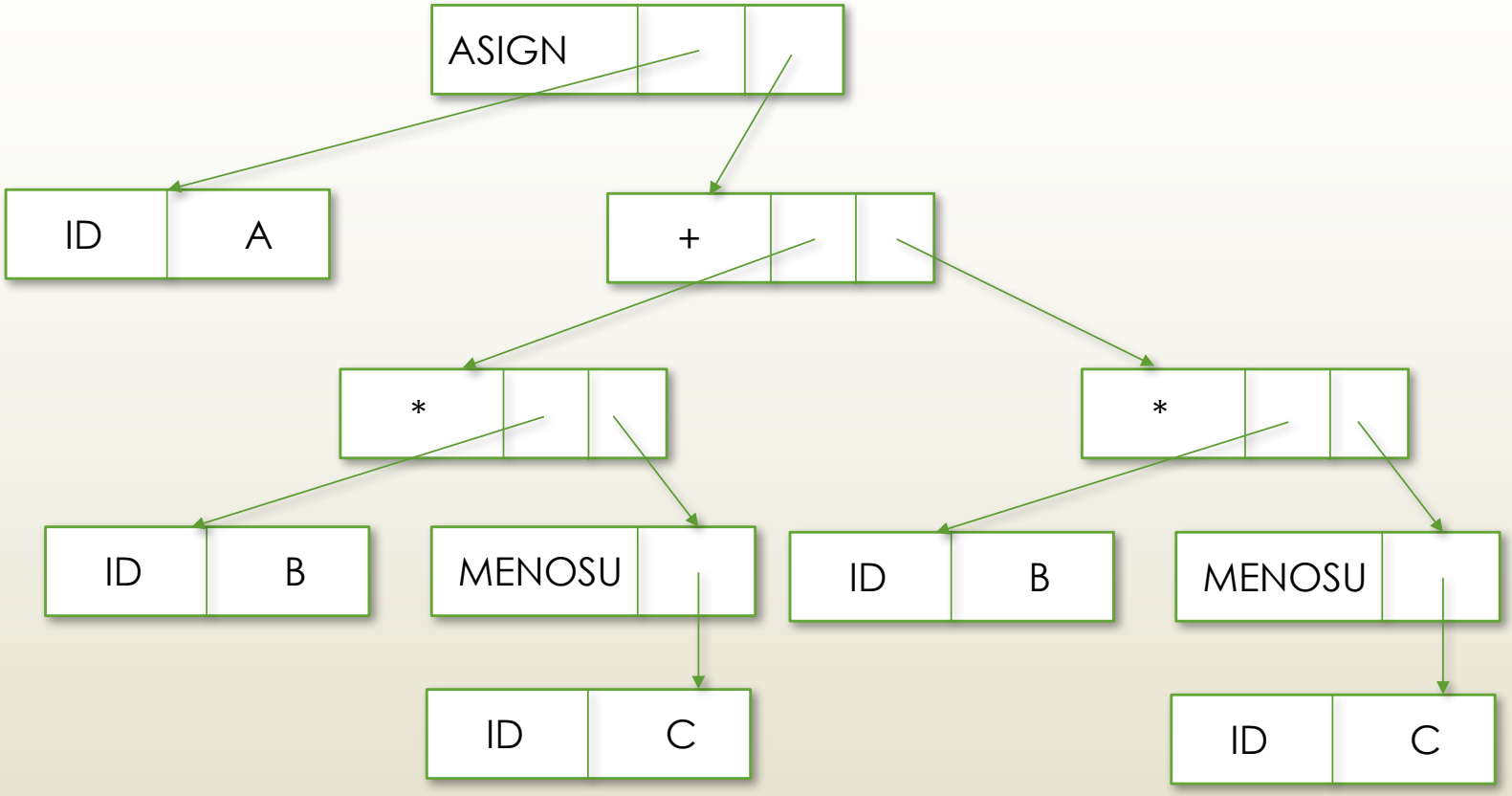
Translation scheme for ST construction

- 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

node.h

```
#ifndef NODE_H
#define NODE_H

#include <string>

namespace Tree {

    class Node{
    public:
        Node(void){}
        virtual void escribe(int level, bool old, bool ne){}
    };

    class InternalNode: public Node{
        char label;
        Node *pn1;
        Node *pn2;
    public:
        InternalNode(int, Node*, Node*);
        void escribe(int level, bool old, bool ne);
    };
```

Example: creation of ST

node.h

```
    class UniqueNode : public Node{
        char label;
        Node* pn1;
    public:
        UniqueNode(int, Node*);
        void escribe(int level, bool old, bool, ne);
    };
    class NodeId: public Node{
        string name;
    public:
        NodeId(string);
        void escribe(int level, bool old, bool, ne);
    };
    class NodeNum: public Node{
        int value;
    public:
        NodeNum(int);
        void escribe(int level, bool old, bool, ne);
    };
    const int Max = 80;
    char* esp(int level, bool old, bool ne);

} //end namespace "Tree"
```

Example: creation of ST

node.h

```
namespace Errors{

    struct TokenError{
        string message;
        int ActToken;
        string lexema;
        TokenError(string msg, int ta, string lex);
    };

    struct NoCuadra{
        int LookedForToken;
        int ActToken;
        string lexema;
        NoCuadra(int t, int ta, string lex);
    };

} // End of namespace "Errors"

#endif
```

Example: creation of ST

node.cpp

```
#include <string>
#include <iostream>
#include <typeinfo>

#include "node.h"
#include "tokens.h"

using namespace Tree;

int yyparse();

Tree::InternalNode::InternalNode(int e, Node* p1, Node* p2){
     label = e;
     pn1 = p1;
     pn2 = p2;
}

Tree::UniqueNode::UniqeNode(int e, Node* p1){
     label = e;
     pn1 = p1;
}
```

Example: creation of ST

node.cpp

```
Tree::NodeId::NodeId(string n){
    name = n;
}

Tree::NodeNum::NodeNum(int v){
    value = v;
};

void Tree::InternalNode::escribe(int level, bool old, bool ne){
    cout << Tree::esp(level, old, ne) << " node (" << label << ")" << endl;
    pn1 -> escribe(level + 2, ne, true);
    pn2 -> escribe(level + 2, ne, false);
}

void Tree::UniqueNode::escribe(int level, bool old, bool ne){
    cout << Tree::esp(level, old, ne) << " node (" << label << ")" << endl;
    pn1 -> escribe(level + 2, ne, true);
}
```

Example: creation of ST

node.cpp

```
void Tree::NodeId::escribe(int level, bool old, bool ne){
     cout << Tree::esp(level, old, ne) << " ID (" << name << ")" << endl;
}
void Tree::NodeNum::escribe(int level, bool old, bool ne){
     cout << Tree::esp(level, old, ne) << " NUM (" << value << ")" << endl;
}
char line[Tree::Max + 1] = {0};
char* Tree::esp(int level, bool old, bool ne){
     int i;
     if(level > 1)
          if (old)
               strcpy(line + (level - 2), "|  ");
          else
               strcpy(line + (level - 2), "   ");
     if(ne)
          strcpy(line + level, "|-->");
     else
          strcpy(line + level, "|L -->");
     return line;
}
```

Example: creation of ST

node.cpp

```
using namespace Errors;

Errors::TokenErrors::TokenErrors(string msg, int ta, string lex){
     message = msg;
     ActToken = ta;
     lexema = lex;
}

Errors::NoCuadra::NoCuadra(int t, int ta, string lex){
     LookedForToken = t;
     ActToken = ta;
     lexema = lex;
}
```

Example: creation of ST

lexico.l

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string>
#include "node.h"
#include "tokens.h"

%}

id   [a-zA-Z_][a-zA-Z0-9_]*

%%

[ \n\t]  ;
":="     return ASIGN;
[0-9]+   {sscanf(yytext, "%d", &yylval.value); return NUMBER;}
{id}     {yylval.name = (char*)malloc(strlen(yytext)+1); strcpy(yylval.name, yytext); return ID;}
.        return *yytext;
```

Example: creation of ST

tokens.h

```
typedef union{
     char *name;
     int value;
     Tree::Node* point;
}YYSTYPE;

#define ID 257
#define NUMBER 258
#define ASIGN 259

extern YYSTYPE yylval;
```

Example: creation of ST

parser.cpp

```
#include <iostream>
#include <string>
#include "node.h"
#include "tokens.h"

YYSTYPE yylval;
using namespace Tree;
using namespace Errors;

void entrada(void);
Tree::Node* E(void);
Tree::Node* Re(Node*);
Tree::Node* T(void)
Tree::Node* Rt(Node*);
Tree::Node* P(void);
Tree::Node* Rp(Node*);
Tree::Node* F(void);

extern int yylex(void);
extern char* yytext;
int ta;
```

```
void cuadra(int t){
    if(ta == t)
        ta = yylex();
    else
        throw Errors::NoCuadra(t, ta, string(yytext));
}
void entrada(void){
    Tree::Node *es, *es1;
    if(ta == ID){
        char* na = yylval.name;
        try{
            cuadra(ID);
            cuadra(ASIGN);
            es = E();
            es1 = new Tree::InternalNode('=', new Tree::NodeId(string(na)), es);
            es1 -> escribe(0,false, false);
        }
```

Example: creation of ST

parser.cpp

```
        catch(Errors::NoCuadra & e){
            cerr << "Error:No cuadra,token que busco = " << e.LookedForToken << "pero hay =
                " << e.ActToken << " (" << e.lexema << ")Salta hasta ;" << endl;
            while((ta==yylex()) != ';' && ta != 0);
        }
        catch(Errorrs::TokenError & e){
            cerr << "Error: " << e.message << "token actual = " << e.ActToken << " (" <<
                    e.lexema << ")Salta hasta ;" << endl;
            while((ta==yylex()) != ';' && ta != 0);
        }
        cuadra(';');
        entrada();
    }
    else if(ta != 0)
        throw Errors::TokenError("en entrada: espera fin de archivo", ta, string(yytext));
}//end of entrada
```

Example: creation of ST

parser.cpp

```
Node* E(void){
     Tree::Node *t_s, *e_s;
     if(ta == NUMBER || ta == ID || ta == '-' || ta == '('){
          t_s = T();
          e_s = Re(t_s);
          return e_s;
     }
     else
          throw Errors::TokenError("en E", ta, string(yytext));
}//end of E

Node* Re(Node* re_h){
     Tree::Node *t_s, *re1_h, *re_s;
     if(ta == '+'){
          cuadra('+');
          t_s = T();
          re1_h = new Tree::InternalNode('+',re_h, t_s);
          re_s = Re(re1_h);
          return re_s;
     }
```

Example: creation of ST

parser.cpp

```
    else if(ta == '-'){
        cuadra('-');
        t_s = T();
        re1_h = new Tree::InternalNode('-',re_h, t_s);
        re_s = Re(re1_h);
        return re_s;
    }
    else
        return re_h;
}//end of Re

Node* T(void){
    Tree::Node *p_s, *t_s;
    if(ta == NUMBER || ta == IND || ta == '-' || ta == '('){
        p_s = P();
        t_s = Rt(p_s);
        return t_s;
    }
    else
        throw Errors::TokenError("En T", ta, string(yytext));
}//end of T
```

Example: creation of ST

parser.cpp

```
Node* Rt(Node* rt_h){
     Tree::Node *p_s, *rt1_h, *rt_s;
     if(ta == '*'){
          cuadra('*');
          p_s = P();
          rt1_h = new Tree::InternalNode('*', rt_h, p_s);
          rt_s = Rt(rt1_h);
          return rt_s;
     }
     else if(ta == '/'){
          cuadra('/');
          p_s = P();
          rt1_h = new Tree::InternalNode('/', rt_h, p_s);
          rt_s = Rt(rt1_h);
          return rt_s;
     }
     else
          return rt_h;
}//end of Rt
```

Example: creation of ST

parser.cpp

```
Node* P(void){
     Tree::Node *p_s, *f_s;
     if(ta == NUMBER || ta == ID || ta == '-' || ta == '('){
          f_s = F();
          p_s = Rp(f_s);
          returs p_s;
     }
     else
          throw Errors::TokenError("En P", ta, string(yytext));
}//end of P

Node* Rp(Node* rp_h){
     Tree::Node *f_s, *rp1_s, *rp_s;
     if(ta == '^'){
          cuadra('^');
          f_s = F();
          rp1_s = Rp(f_s);
          rp_s = new Tree::InternalNode('^', rp_h, rp1_s);
          return rp_s;
     }else
          return rp_h;
}//end of Rp
```

Example: creation of ST

parser.cpp

```
Node* F(void){
     Tree::Node * f_s; *f1_s;
     if(ta == NUMBER){
          f_s = new Tree::NodeNum(yylval.value);
          cuadra(NUMBER);
          retur f_s;
     }else if (ta == ID){
          f_s = new Tree::NudeId(yylval.name);
          cuadra(ID);
          return f_s;
     }else if(ta == '('){
          cuadra('(');
          f_s = E();
          cuadra(')');
          return f_s;
     }else if(ta == '-'){
          cuadra('-');
          f1_s = F();
          f_s = new Tree::UniqueNode('~',f1_s);
          return f_s;
     }
     else
          throw Errors::TokenError("En F", ta, string(yytext));
}// end of F
```

Example: creation of ST

parser.cpp

```
int main(){
     cout << "Comienza el programa" << endl;
     cout << "Teclee expresiones aritmeticas terminadas en punto y coma" << endl;
     ta = yylex();
     try{
          entrada();
     }
     catch(Errors::NoCuadra & e){
          cerr << "Error:No cuadra,token que busco = " << e.LookedForToken << "pero hay =
                  " << e.ActToken << " (" << e.lexema << ")Fin del programa" << endl;
          exit(EXIT_FAILURE);
     }
     catch(Errorrs::TokenError & e){
          cerr << "Error: " << e.message << "token actual = " << e.ActToken << " (" <<
                  e.lexema << ")Final del programa" << endl;
          exit(EXIT_FAILURE);
     }
     cout << "Fin del programa" << endl;
}//end of main()
```