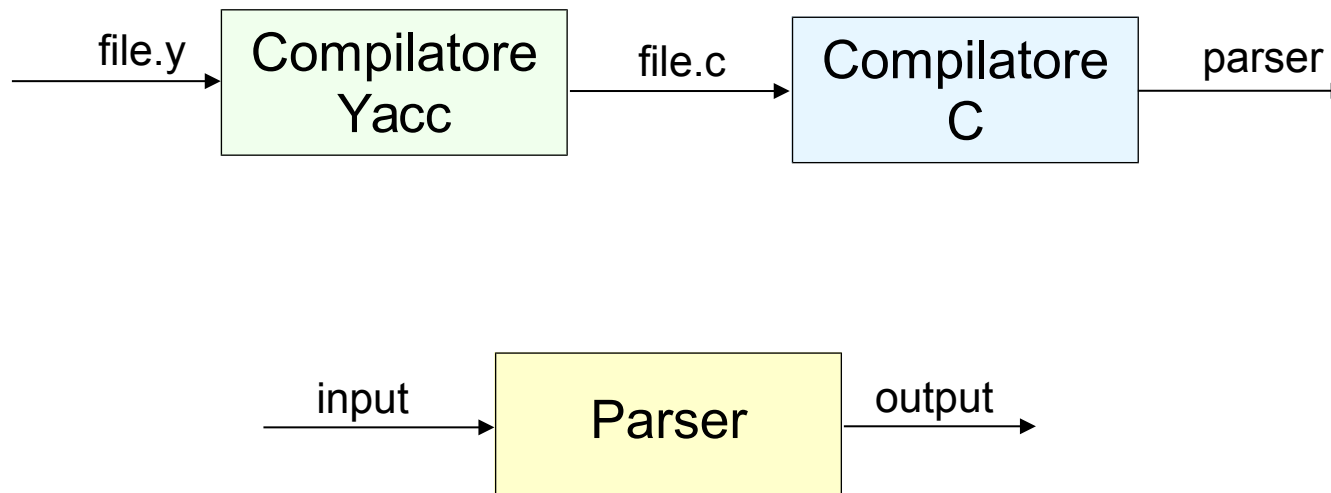


# Yacc

- Generatore di parser LALR(1)
- YACC = “*Yet Another Compiler Compiler*” → sintomo di due fatti:
  1. Popularità dei generatori di parser in quegli anni
  2. Storicamente: fasi del compilatore intrecciate con l'analisi sintattica



# Yacc (ii)

- Specifica Yacc: strutturalmente identica a Lex

*Dichiarazioni*

%%

*Regole di traduzione*

%%

*Funzioni ausiliarie*

- Dichiarazioni  $\left\{ \begin{array}{l} \text{black box (definizioni ausiliarie): \% \{ \#include, costanti, variabili \% \}} \\ \text{white box (token, ... )} \end{array} \right.$

- Esempio: calcolatore (interprete)

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow ( E ) \mid \text{digit} \end{array}$$


Ricorsiva a sinistra

# Yacc (iii)

$L \rightarrow E \text{ eol}$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow ( E ) \mid \text{digit}$

```

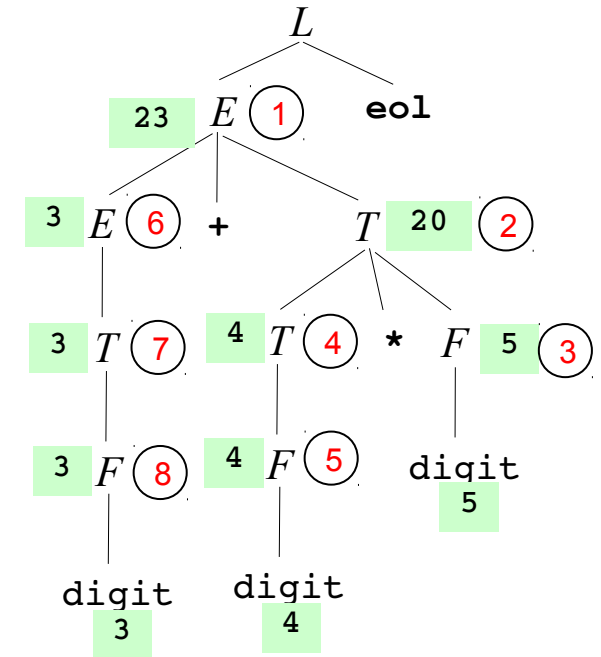
%{
#include <stdio.h>
#include <ctype.h>
int yylex();
void yyerror();
}%
%token DIGIT
%%
line      :   expr '\n'   { printf("%d\n", $1); }
           ;
expr      :   expr '+' term { $$ = $1 + $3; }
           |   term
           ;
term      :   term '*' factor { $$ = $1 * $3; }
           |   factor
           ;
factor    :   '(' expr ')' { $$ = $2; }
           |   DIGIT
           ;
%%
int yylex()
{ int c;
  c = getchar();
  if (isdigit(c)){
    yylval = c - '0';
    return(DIGIT);
  }
  return(c);
}

void yyerror(){fprintf(stderr, "Syntax error\n");}

void main(){yyparse();}
    
```

valore lessicale ←

3 + 4 \* 5



# Yacc (iv)

1. **Dichiarazioni**  $\left\langle \begin{array}{l} \% \{ \text{dichiarazioni C } \% \} \\ \text{dichiarazioni di terminali (token) di G} \end{array} \right.$

`%token DIGIT`  $\Rightarrow$  `#define DIGIT 258`

2. **Regole di traduzione** = regole di produzione + azioni semantiche

$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$

$\Rightarrow$

```
A :  $\alpha_1$  { azione 1 }
   |  $\alpha_2$  { azione 2 }
   ...
   |  $\alpha_n$  { azione n }
   ;
```

- Assioma = primo nonterminale (default), o `%start line`
- 2 modi per riconoscere token  $\left\langle \begin{array}{l} '+' \\ \text{DIGIT} \end{array} \right.$
- 'c' = simbolo terminale 'c'
- Nonterminale = stringa di caratteri alfanumerici
- Alternative separate da **|**
- Separazione do ogni gruppo di alternative + azioni semantiche da **;**
- Azione semantica = frammento di codice C
- **Pseudo-variabili** per referenziare valori di **attributi semantici** (default: intero)  $\left\langle \begin{array}{l} \$\$ : \text{sinistra} \\ \$i : \text{i-esimo destra} \end{array} \right.$

# Yacc (v)

- `yylval` = variabile contenente il valore lessicale dei token → assegnata dal lexer (valore associato al terminale spostato sulla pila)
- Azione semantica eseguita nella riduzione `$$ = f($1, $2, ...)`

```
expr  : expr '+' term { $$ = $1 + $3; }  
      | term  
      ;
```

azione di default: `$$ = $1;`

### 3. **Funzioni ausiliarie** = funzioni C necessarie per completare la funzione di parsing

In particolare  $\begin{cases} \text{yylex}() \\ \text{yyerror}() \end{cases} \Rightarrow$  chiamate da `yyparse()` → return  $\begin{cases} 0: \text{ok} \\ 1: \text{errore} \end{cases}$

# Yacc (vi)

- G ambigua → conflitti → individuati da Yacc (opzione `-v` : mostra anche le soluzioni)
- Se  $\exists$  conflitti → consultare `file.output` per vedere  $\left\{ \begin{array}{l} \text{conflitti} \\ \text{soluzioni} \end{array} \right.$
- Regole Yacc per risoluzione dei conflitti:
  1. Spostamento/riduzione → scelto lo spostamento
  2. Riduzione/riduzione → scelta la prima regola di produzione (nel file)

# Yacc (vii)

Opzioni di Yacc:

**-d** (header): genera `file.h` = dichiarazioni delle informazioni esportabili

```
#define YYSTYPE int  
  
Symboli per Lex  
  
extern YYSTYPE yylval;
```

**-v** (verbose): genera `file.output` = descrizione della tabella di parsing LALR(1)

Pragmaticamente: **Prima**: eseguire Yacc su G (senza azioni semantiche, funzioni ausiliarie, ...) per essere sicuri che il parser generato sia conforme alle aspettative  
**Poi**: completamento

**YYDEBUG** : Traccia della esecuzione del parser generato da Yacc (non traccia di Yacc!)

Simbolo che deve essere definito, tipicamente opzione **-D** del compilatore C : **-DYYDEBUG**

Effettiva tracciatura: abilitata dalla variabile intera `yydebug`:

```
extern int yydebug;  
yydebug = 1;
```

⇒ lista delle azioni eseguite dal parser per un input

# Yacc (viii)

- Generalizzazione del tipo di valori computati dalle azioni semantiche (cioè: tipo delle pseudo-variabili, es. calcolatore per numeri reali)

```
%{  
...  
#define YYSTYPE float  
...  
%}
```

- Definizione del tipo in un file separato: `typedef ... TYPE;`

```
#define YYSTYPE TYPE
```

 (nel file Yacc)

Esempio: Construzione dell'albero sintattico: `typedef ... *PNODE;`

↓  
puntatore al nodo dell'albero



# Yacc (ix)

- Azioni semantiche **embedded**: quando necessario eseguire codice prima del riconoscimento completo di una produzione

```
decl → type var-list ;
type → int | float
var-list → id , var-list | id
```

```
int a, b, c;
```

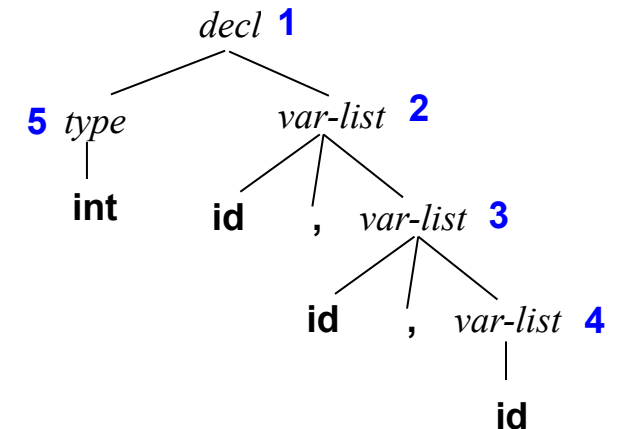
Goal: Analizzando gli identificatori in *var-list*, qualificare ogni **id** con il rispettivo tipo.

```
decl      :   type var-list ';'
          ;

type      :   INT      {current_type = INT_TYPE;}
          |   FLOAT    {current_type = FLOAT_TYPE;}
          ;

var-list  :   ID {insert(yytext, current_type);} ',' var-list
          |   ID {insert(yytext, current_type);}
          ;
```

variabile statica



- Interpretazione delle azioni embedded da parte di Yacc:

```
A : B { azione embedded } C;
```

≡

```
A : B E C;
E : { azione embedded };
```

$\epsilon$ -produzione

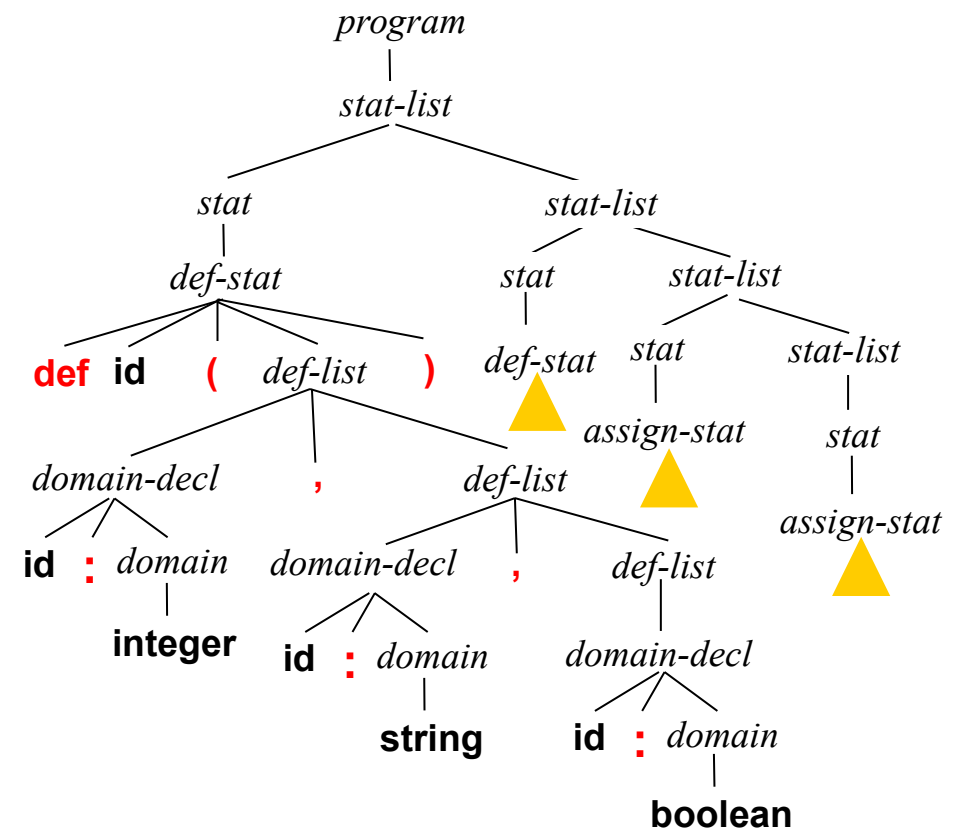
# Costruzione Bottom-up dell'Albero (Semi) Concreto

```

program  $\rightarrow$  stat-list
stat-list  $\rightarrow$  stat stat-list | stat
stat  $\rightarrow$  def-stat | assign-stat
def-stat  $\rightarrow$  def id ( def-list )
def-list  $\rightarrow$  domain-decl , def-list | domain-decl
domain-decl  $\rightarrow$  id : domain
domain  $\rightarrow$  integer | string | boolean
assign-stat  $\rightarrow$  id := { tuple-list }
tuple-list  $\rightarrow$  tuple-const tuple-list |  $\epsilon$ 
tuple-const  $\rightarrow$  ( simple-const-list )
simple-const-list  $\rightarrow$  simple-const , simple-const-list | simple-const
simple-const  $\rightarrow$  intconst | strconst | boolconst

```

```
def R (A: integer, B: string, C: boolean)
def S (D: integer, E: string)
R := {(3, "alpha", true)(5, "beta", false)}
S := {(125, "sun")(236, "moon")}
```



# def.h

```
#include <stdio.h>
#include <stdlib.h>

typedef enum
{
    NPROGRAM,
    NSTAT_LIST,
    NSTAT,
    NDEF_STAT,
    NDEF_LIST,
    NDOMAIN_DECL,
    NDOMAIN,
    NASSIGN_STAT,
    NTUPLE_LIST,
    NTUPLE_CONST,
    NSIMPLE_CONST_LIST,
    NSIMPLE_CONST
} Nonterminal;

typedef enum
{
    T_INTEGER,
    T_STRING,
    T_BOOLEAN,
    T_INTCONST,
    T_BOOLCONST,
    T_STRCONST,
    T_ID,
    T_NONTERMINAL
} Typenode;
```

```
typedef union
{
    int ival;
    char *sval;
    enum {FALSE, TRUE} bval;
} Value;

typedef struct snode
{
    Typenode type;
    Value value;
    struct snode *child, *brother;
} Node;

typedef Node *Pnode;
```

```
char *newstring(char*);

int yylex();

Pnode nontermnode(Nonterminal),
    idnode(),
    keynode(Typenode),
    intconstnode(),
    strconstnode(),
    boolconstnode(),
    newnode(Typenode);

void treeprint(Pnode, int),
    yyerror();
```

# lexer.lex

```
%{
#include "parser.h"
#include "def.h"
int line = 1;
Value lexval;
}%
%option noyywrap

spacing      ([ \t])+
letter       [A-Za-z]
digit        [0-9]
intconst     {digit}+
strconst     \"([^\"])*\"
boolconst    false|true
id           {letter}({letter}|{digit})*
sugar        [( ){:},]
%%

{spacing}    ;
\n           {line++;}
def          {return(DEF);}
integer      {return(INTEGER);}
string       {return(STRING);}
boolean      {return(BOOLEAN);}
{intconst}   {lexval.ival = atoi(yytext); return(INTCONST);}
{strconst}   {lexval.sval = newstring(yytext); return(STRCONST);}
{boolconst}  {lexval.bval = (yytext[0] == 'f' ? FALSE : TRUE);
              return(BOOLCONST);}
{id}         {lexval.sval = newstring(yytext); return(ID);}
{sugar}      {return(yytext[0]);}
":="        {return(ASSIGN);}
.           {return(ERROR);}
%%
```

```
char *newstring(char *s)
{
    char *p;

    p = malloc(strlen(s)+1);
    strcpy(p, s);
    return(p);
}
```

# parser.h

```
#define DEF          258
#define INTEGER      259
#define STRING       260
#define BOOLEAN      261
#define ID           262
#define INTCONST     263
#define STRCONST     264
#define BOOLCONST    265
#define ASSIGN       266
#define ERROR        267
```

# parser.y

```
%{
#include "def.h"
#define YYSTYPE Pnode
extern char *yytext;
extern Value lexval;
extern int line;
extern FILE *yyin;
Pnode root = NULL;
%}

%token DEF INTEGER STRING BOOLEAN ID INTCONST STRCONST BOOLCONST ASSIGN
%token ERROR

%%

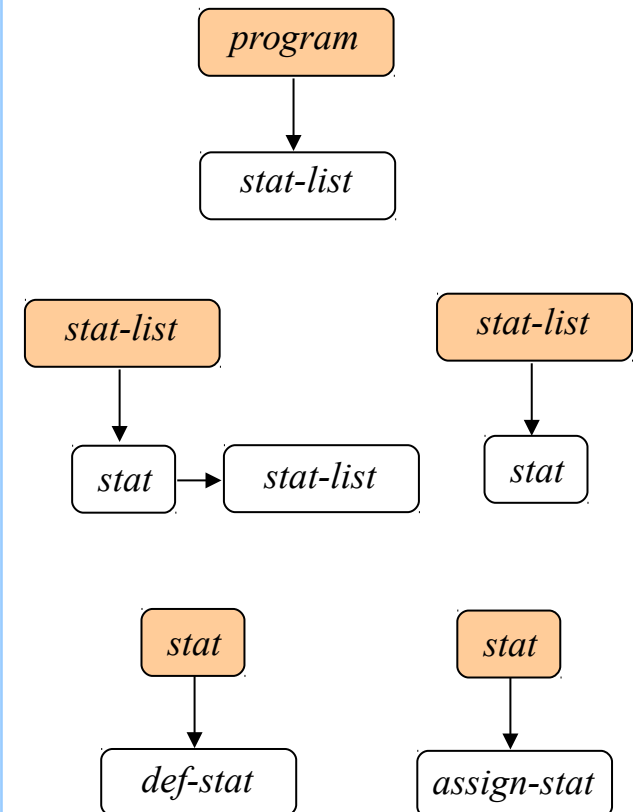
program : stat_list {root = $$ = nontermnode(NPROGRAM);
                    $$->child = $1;}
        ;

stat_list : stat stat_list {$$ = nontermnode(NSTAT_LIST);
                          $$->child = $1;
                          $1->brother = $2;}
        | stat {$$ = nontermnode(NSTAT_LIST);
               $$->child = $1;}
        ;

stat : def_stat {$$ = nontermnode(NSTAT);
           $$->child = $1;}
     | assign_stat {$$ = nontermnode(NSTAT);
                   $$->child = $1;}
     ;
```

} analizzatore lessicale

*program* → *stat-list*  
*stat-list* → *stat stat-list* | *stat*  
*stat* → *def-stat* | *assign-stat*



# parser.y (ii)

*def-stat* → **def id ( def-list )**

*def-list* → *def-list, domain-decl* | *domain-decl*

*domain-decl* → **id : domain**

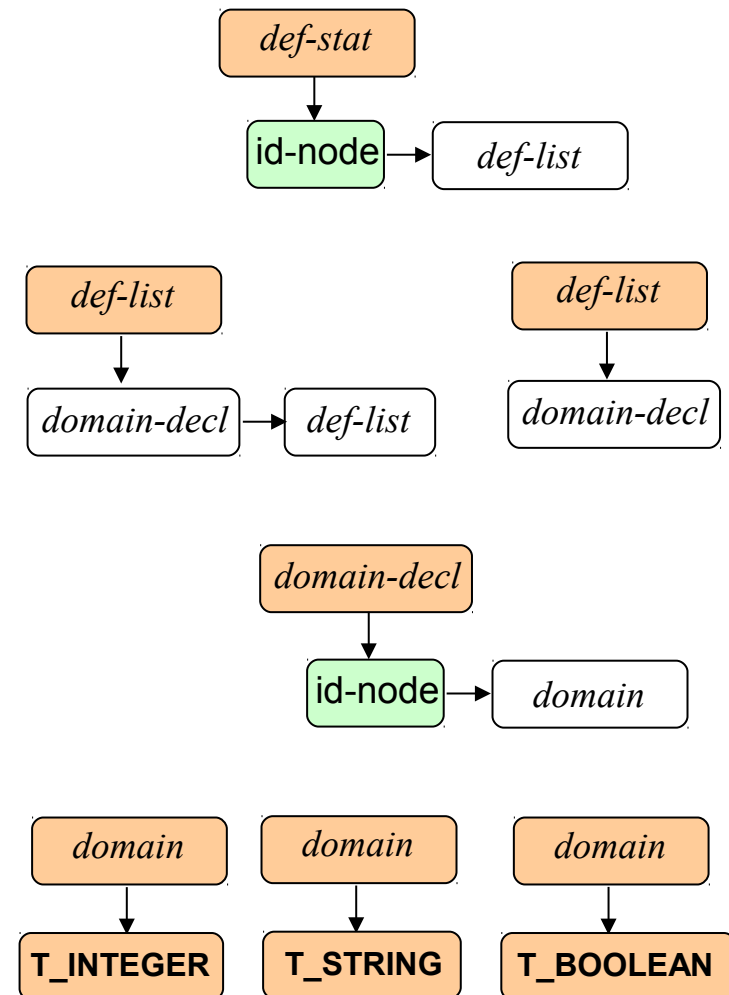
*domain* → **integer** | **string** | **boolean**

```
def_stat : DEF
        ID { $$ = idnode(); }
        '(' def_list ')' { $$ = nontermnode(NDEF_STAT);
                          $$->child = $3;
                          $3->brother = $5; }
        ;

def_list : domain_decl ',' def_list { $$ = nontermnode(NDEF_LIST);
                                     $$->child = $1;
                                     $1->brother = $3; }
        | domain_decl { $$ = nontermnode(NDEF_LIST);
                       $$->child = $1; }
        ;

domain_decl : ID { $$ = idnode(); }
            ':' domain { $$ = nontermnode(NDOMAIN_DECL);
                      $$->child = $2;
                      $2->brother = $4; }
            ;

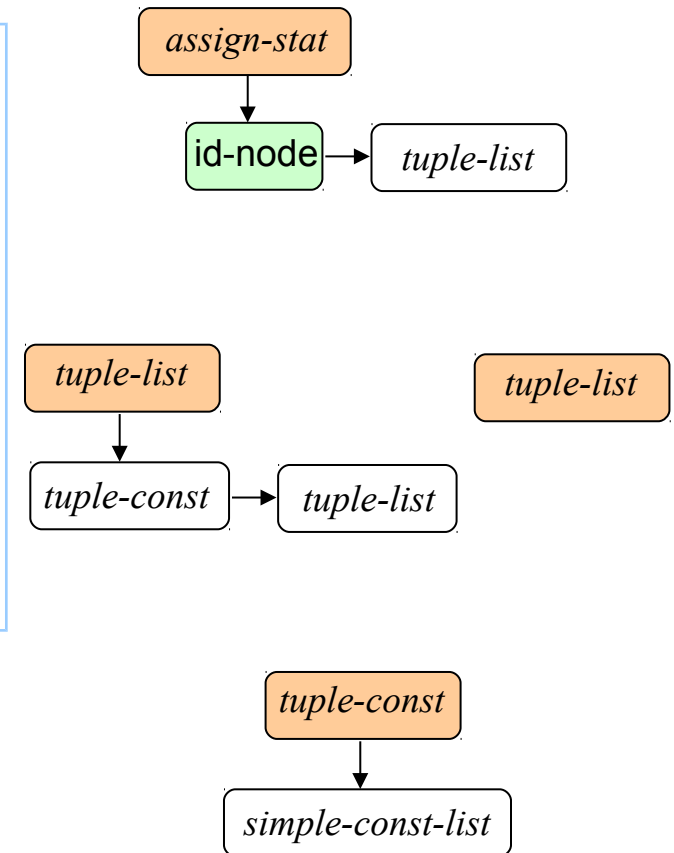
domain : INTEGER { $$ = nontermnode(NDOMAIN);
                 $$->child = keynode(T_INTEGER); }
        | STRING { $$ = nontermnode(NDOMAIN);
                  $$->child = keynode(T_STRING); }
        | BOOLEAN { $$ = nontermnode(NDOMAIN);
                    $$->child = keynode(T_BOOLEAN); }
        ;
```



# parser.y (iii)

*assign-stat* → **id** := { *tuple-list* }  
*tuple-list* → *tuple-const* *tuple-list* | ε  
*tuple-const* → ( *simple-const-list* )

```
assign_stat : ID {$$ = idnode();}  
            ASSIGN '{' tuple_list '}' {$$ = nontermnode(NASSIGN_STAT);  
                                     $$->child = $2;  
                                     $2->brother = $5;}  
            ;  
  
tuple_list : tuple_const tuple_list {$$ = nontermnode(NTUPLE_LIST);  
                                     $$->child = $1;  
                                     $1->brother = $2;}  
            | {$$ = nontermnode(NTUPLE_LIST);}  
            ;  
  
tuple_const : '(' simple_const_list ')' {$$ = nontermnode(NTUPLE_CONST);  
                                         $$->child = $2;}  
            ;
```

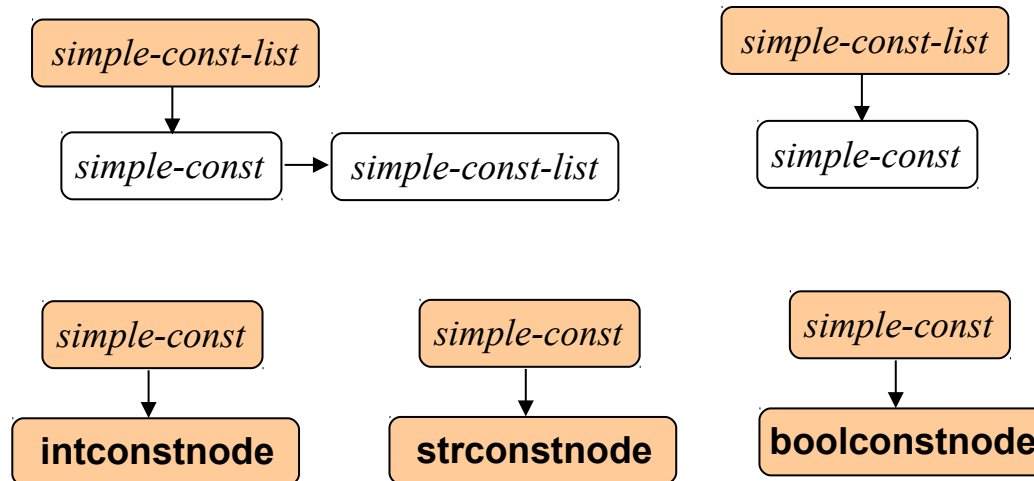




# parser.y (iv)

*simple-const-list* → *simple-const* , *simple-const-list* | *simple-const*  
*simple-const* → **intconst** | **strconst** | **boolconst**

```
simple_const_list : simple_const ',' simple_const_list {$$ = nontermnode(NSIMPLE_CONST_LIST);  
                                                         $$->child = $1;  
                                                         $1->brother = $3;}  
                | simple_const {$$ = nontermnode(NSIMPLE_CONST_LIST);  
                                                         $$->child = $1;}  
                ;  
  
simple_const : INTCONST {$$ = nontermnode(NSIMPLE_CONST); $$->child = intconstnode();}  
            | STRCONST {$$ = nontermnode(NSIMPLE_CONST); $$->child = strconstnode();}  
            | BOOLCONST {$$ = nontermnode(NSIMPLE_CONST); $$->child = boolconstnode();}  
            ;  
  
%%
```



## parser.y (v)

```
Pnode nontermnode(Nonterminal nonterm)
{
    Pnode p = newnode(T_NONTERMINAL);
    p->value.ival = nonterm;
    return(p);
}
```

```
Pnode idnode()
{
    Pnode p = newnode(T_ID);
    p->value.sval = lexval.sval;
    return(p);
}
```

```
Pnode keynode(Typenode keyword)
{
    return(newnode(keyword));
}
```

```
Pnode intconstnode()
{
    Pnode p = newnode(T_INTCONST);
    p->value.ival = lexval.ival;
    return(p);
}
```

```
Pnode strconstnode()
{
    Pnode p = newnode(T_STRCONST);
    p->value.sval = lexval.sval;
    return(p);
}
```

```
Pnode boolconstnode()
{
    Pnode p = newnode(T_BOOLCONST);
    p->value.bval = lexval.bval;
    return(p);
}
```

```
Pnode newnode(Typenode tnode)
{
    Pnode p = malloc(sizeof(Node));
    p->type = tnode;
    p->child = p->brother = NULL;
    return(p);
}
```

```
int main()
{
    int result;

    yyin = stdin;
    if((result = yyparse()) == 0)
        treeprint(root, 0);
    return(result);
}
```

```
void yyerror()
{
    fprintf(stderr, "Line %d: syntax error on symbol \"%s\"\n",
            line, yytext);
    exit(-1);
}
```

# makefile

```
bup: lexer.o parser.o tree.o
    cc -g -o bup lexer.o parser.o tree.o

lexer.o: lexer.c parser.h def.h
    cc -g -c lexer.c

parser.o: parser.c def.h parser.dot
    cc -g -c parser.c
    dot -Tpdf -o parser.pdf parser.dot

tree.o: tree.c def.h
    cc -g -c tree.c

lexer.c: lexer.lex parser.y parser.h parser.c def.h
    flex -o lexer.c lexer.lex

parser.h: parser.y def.h
    bison -vdg -o parser.c parser.y
```

# Costruzione Bottom-up dell'Albero Astratto

```
program → stat-list  
stat-list → stat stat-list | stat  
stat → def-stat | assign-stat  
def-stat → def id ( def-list )  
def-list → domain-decl , def-list | domain-decl  
domain-decl → id : domain  
domain → integer | string | boolean  
assign-stat → id := { tuple-list }  
tuple-list → tuple-const tuple-list | ε  
tuple-const → ( simple-const-list )  
simple-const-list → simple-const , simple-const-list | simple-const  
simple-const → intconst | strconst | boolconst
```

```
program → stat { stat }  
stat → def-stat | assign-stat  
def-stat → def id ( def-list )  
def-list → id : domain { , id: domain }  
domain → integer | string | boolean  
assign-stat → id := { { tuple-const } }  
tuple-const → ( simple-const { , simple-const } )  
simple-const → intconst | strconst | boolconst
```

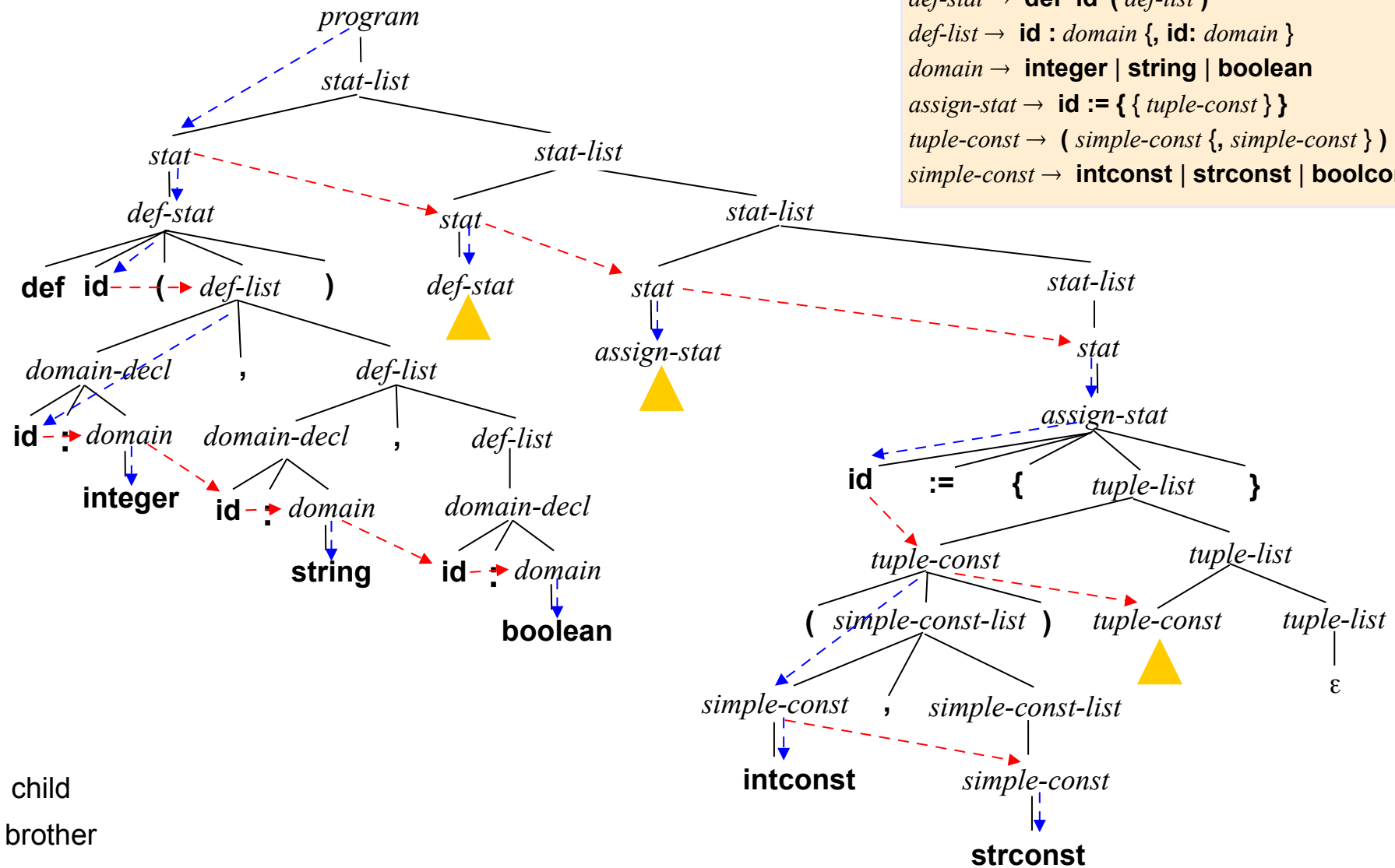
```
def R (A: integer, B: string, C: boolean)  
def S (D: integer, E: string)  
R := {(3, "alpha", true)(5, "beta", false)}  
S := {(125, "sun")(236, "moon")}
```

## Costruzione Bottom-up dell'Albero Astratto (ii)

```

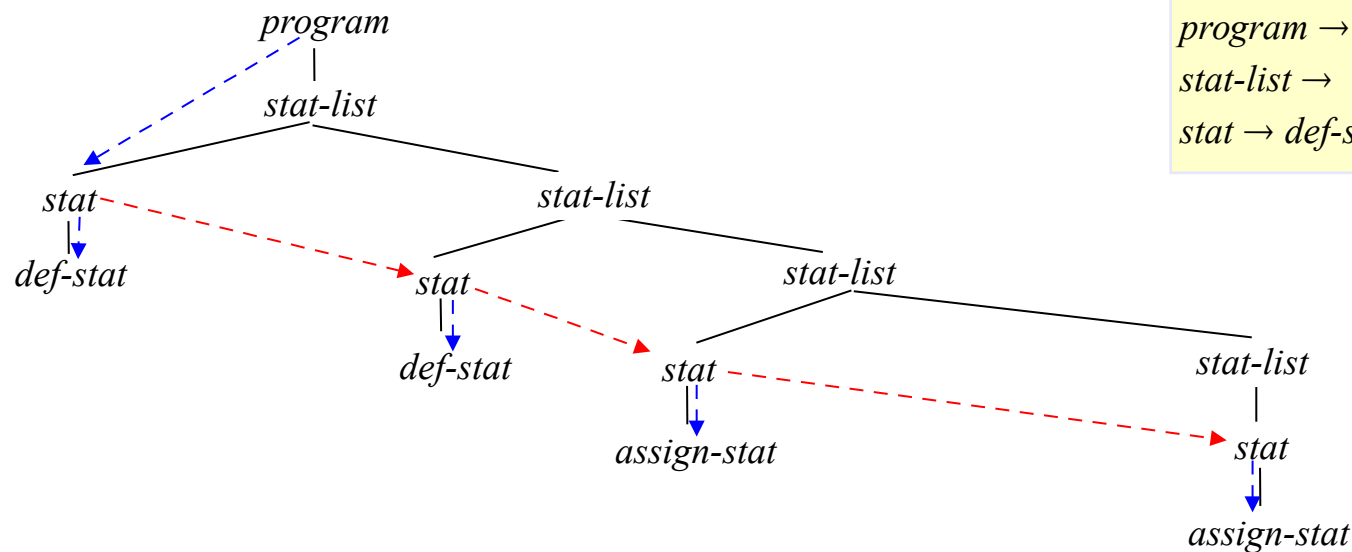
program → stat { stat }
stat → def-stat | assign-stat
def-stat → def id ( def-list )
def-list → id : domain { , id: domain }
domain → integer | string | boolean
assign-stat → id := { { tuple-const } }
tuple-const → ( simple-const { , simple-const } )
simple-const → intconst | strconst | boolconst

```



- child
- brother

# Costruzione Bottom-up dell'Albero Astratto (iii)



$program \rightarrow stat\_list$   
 $stat\_list \rightarrow stat\ stat\_list \mid stat$   
 $stat \rightarrow def\_stat \mid assign\_stat$

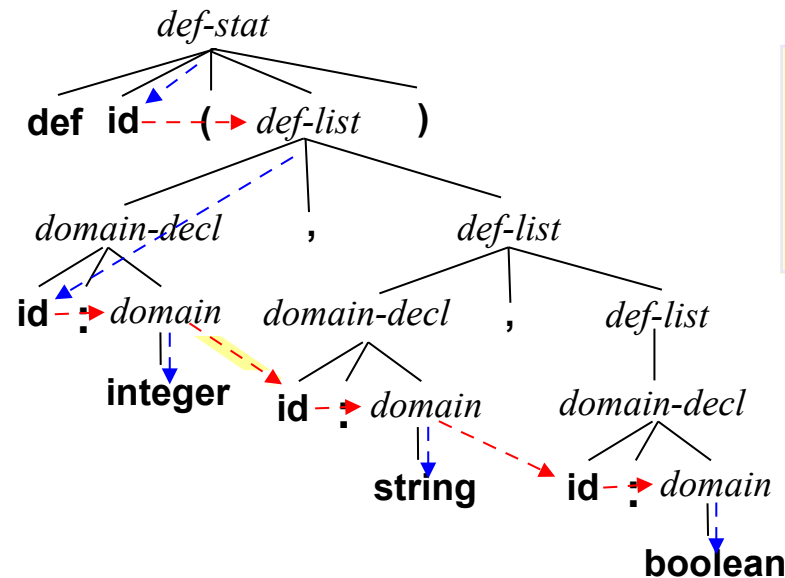
```

program : stat_list {root = $$ = nontermnode(NPROGRAM); $$->child = $1;}
        ;

stat_list : stat stat_list {$$ = $1; $1->brother = $2}
          | stat {$$ = $1;}
          ;

stat : def_stat {$$ = nontermnode(NSTAT); $$->child = $1;}
     | assign_stat {$$ = nontermnode(NSTAT); $$->child = $1;}
     ;
    
```

# Costruzione Bottom-up dell'Albero Astratto (iv)



$def-stat \rightarrow \mathbf{def\ id\ (}\ def-list \mathbf{)}$   
 $def-list \rightarrow domain-decl\ ,\ def-list\ |\ domain-decl$   
 $domain-decl \rightarrow \mathbf{id\ :}\ domain$   
 $domain \rightarrow \mathbf{integer\ |\ string\ |\ boolean}$

```

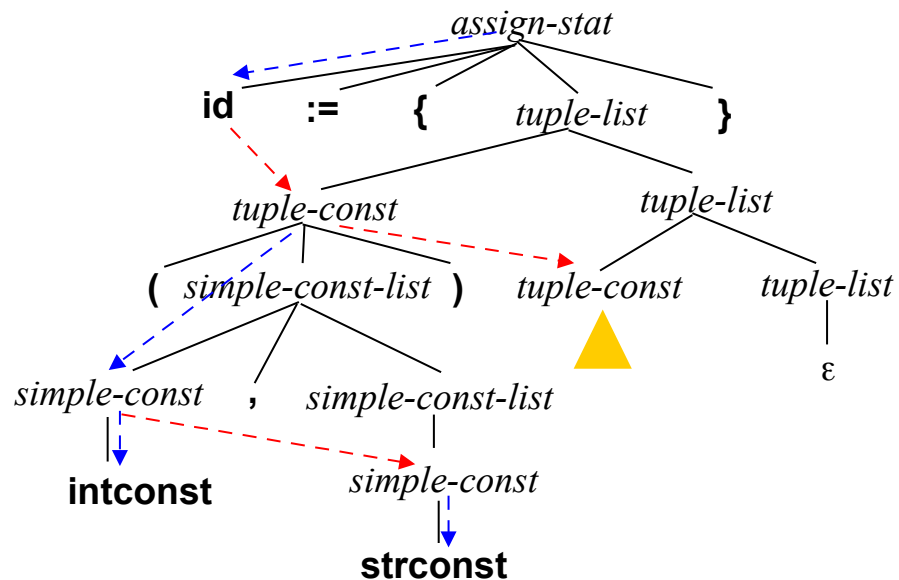
def_stat : DEF ID {$$ = idnode();} '(' def_list ')' {$$ = nontermnode(NDEF_STAT);
                                                $$->child = $3;
                                                $3->brother = nontermnode(NDEF_LIST);
                                                $3->brother->child = $5;}
;

def_list : domain_decl ',' def_list {$$ = $1; $1->brother->brother = $3;}
        | domain_decl {$$ = $1;}
;

domain_decl : ID {$$ = idnode();} ':' domain {$$ = $2; $2->brother = $4;}
;

domain : INTEGER {$$ = nontermnode(NDOMAIN); $$->child = keynode(T_INTEGER);}
        | STRING {$$ = nontermnode(NDOMAIN); $$->child = keynode(T_STRING);}
        | BOOLEAN {$$ = nontermnode(NDOMAIN); $$->child = keynode(T_BOOLEAN);}
;
    
```

# Costruzione Bottom-up dell'Albero Astratto (v)



$assign-stat \rightarrow id := \{ tuple-list \}$   
 $tuple-list \rightarrow tuple-const tuple-list \mid \epsilon$   
 $tuple-const \rightarrow ( simple-const-list )$   
 $simple-const-list \rightarrow simple-const , simple-const-list \mid simple-const$   
 $simple-const \rightarrow intconst \mid strconst \mid boolconst$

```

assign_stat : ID { $$ = idnode(); } ASSIGN '{' tuple_list '}' { $$ = nontermnode(NASSIGN_STAT);
                                                    $$->child = $2; $2->brother = $5; }
;

tuple_list : tuple_const tuple_list { $$ = $1; $1->brother = $2; }
           | { $$ = NULL; }
;

tuple_const : '(' simple_const_list ')' { $$ = nontermnode(NTUPLE_CONST); $$->child = $2; }
;

simple_const_list : simple_const ',' simple_const_list { $$ = $1; $1->brother = $3; }
                 | simple_const { $$ = $1; }
;

simple_const : INTCONST { $$ = nontermnode(NSIMPLE_CONST); $$->child = intconstnode(); }
            | STRCONST { $$ = nontermnode(NSIMPLE_CONST); $$->child = strconstnode(); }
            | BOOLCONST { $$ = nontermnode(NSIMPLE_CONST); $$->child = boolconstnode(); }
;
    
```