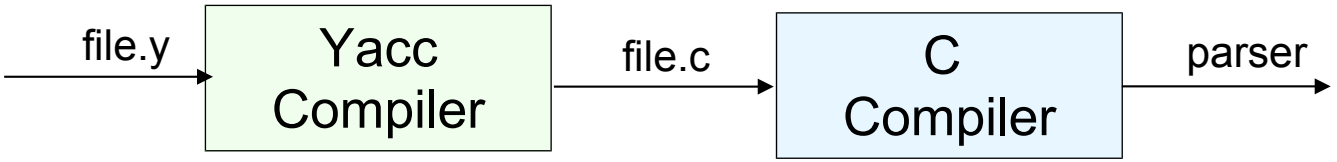
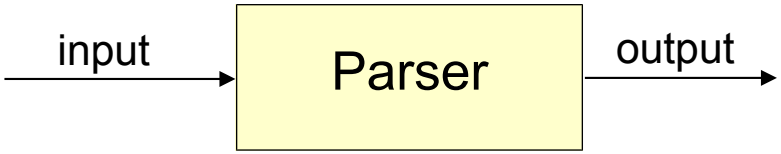
Yacc

- Generator of LALR(1) parsers
- YACC = "*Yet Another Compiler Compiler*" $\rightarrow$ symptom of two facts:
  1. Popularity of parser generators in the '70s
  2. Historically: compiler phases mixed within syntax analysis





Yacc (ii)

- Yacc specification: structurally identical to Lex

Declarations
%%
Translation rules
%%
Auxiliary functions

- Declarations
  - black box (**auxiliary definitions**): **%{** #include, constants, variables **%}**
  - white box (tokens, rules of precedence/associativity (for conflict resolution), ...)

- Example: calculator (interpreter)

$E \rightarrow E$ **+** $T \mid T$
$T \rightarrow T$ ***** $F \mid F$
$F \rightarrow$ **(** E **)** | **digit**

⇨ Left recursive

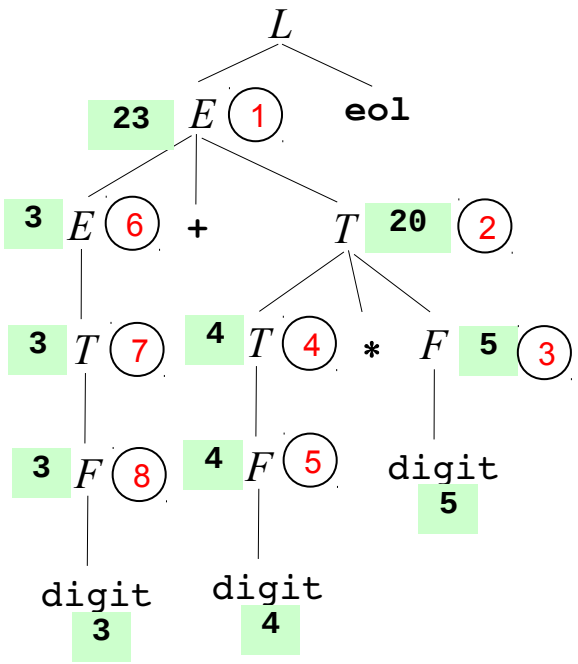
Yacc (iii)

$L \rightarrow E$ **eol**
$E \rightarrow E$ **+** $T \mid T$
$T \rightarrow T$ ***** $F \mid F$
$F \rightarrow$ **(** E **) | digit**

```
%{
#include <stdio.h>
#include <ctype.h>
%}
%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
        ;
%%
yylex()
{ int c;

  c = getchar();
  if (isdigit(c)){
    yylval = c - '0';
    return(DIGIT);
  }
  return(c);
}

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

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

lexical attribute

3 + 4 * 5



Yacc (iv)

1. **Declarations** < `%{` C declarations `%}`
   declaration of terminals (tokens) of G

   `%token DIGIT` ⇨ `#define DIGIT 258`

2. **Translation rules** = production rules + semantic actions

$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ ⇨

```
A  :  α1  { action 1 }
   |  α2  { action 2 }
   ...
   |  αn  { action n }
   ;
```

- Axiom = first nonterminal (default), or `%start line`
- 2 ways for token recognition < `'+'` / `DIGIT`
- 'c' = terminal symbol 'c'
- Nonterminal = string of alphanumeric characters
- Alternatives separated by **|**
- Separation of each group of alternatives + semantic actions by **;**
- Semantic action = fragment of C code
- Pseudo-variables to reference values of semantic attributes (default: integer) < `$$`: left / `$i`: i-th right

Yacc (v)

- `yylval` = variable containing lexical value of tokens → assigned by lexical analyzer
  (value associated with terminal shifted onto the stack)
- Semantic action executed at a reduction `$$ = f($1, $2, ...)`

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

default action: `$$ = $1;`

3. **Auxiliary functions** = C functions necessary for completing the parsing function

In particular: { `yylex()` , `yyerror()` } ⇨ called by `yyparse()` → return { 0: ok , 1: error }

Yacc (vi)

- G ambiguous → conflicts → detected by Yacc (option `-v` : shows the solutions too)
- If ∃ conflicts → consult `file.output` to view 〈 conflicts, solutions
- Yacc rules for conflict resolutions:

  1. Shift/reduce → chosen the shift
  2. Reduce/reduce → chosen the first production (in file)

- Change of default of resolution for conflicts shift/reduce 〈 precedence, associativity

```
%left '+' '-'
%right '^'
%nonassoc '<'
```

`%left '+' '-'` → same precedence, left associative

`%right '^'` → right associative

`%nonassoc '<'` → non-associative (binary) operator — `a < b < c` ← no!

⇩

increasing precedence:

```
%left '+' '-'
%left '*' '/'
```

Yacc (vii)

- Yacc implicitly defines a { precedence / associativity } for production $A \rightarrow \alpha$

  ⇨ that of the rightmost terminal

Rule: Given the choice between { shift **a** / reduce $A \rightarrow \alpha$ } in a shift/reduce conflict:

```
if precedence (A → α) > precedence(a)  or
  (precedence (A → α) = precedence(a) and associativity(A → α) = left) then
       Reduce A → α
else Shift a
```

```
a * b + c
a + b + c
a + b * c
```

Example: $E \rightarrow E + T$. Conflict { shift **+** / reduce $E \rightarrow E + T$ } ← chosen; { shift ***** / reduce $E \rightarrow E + T$ }

- Changing the precedence-default for a production (inherited from rightmost terminal):

`%prec <terminal>` ⇨ added to production

```
expr  :  '-' expr %prec UMINUS   {$$ = -$2;}
```

defined in declarations

Yacc (viii)

Extended calculator

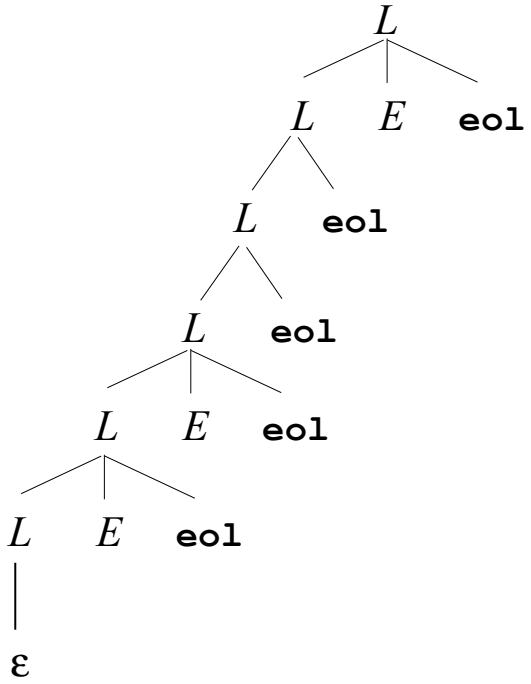
1. List of expressions (one for each line)
2. Possible empty lines between different expressions
3. Spacing within expressions
4. Operator – both unary and binary
5. Numbers with several digits

$L \rightarrow L\ E$ **eol** $\mid L$ **eol** $\mid \varepsilon$

$E \rightarrow E$ **+** $E \mid E$ **–** $E \mid E$ ***** $E \mid E$ **/** $E \mid$ **(** E **)** $\mid$ **–** $E \mid$ **num**

ambiguous

```
E eol E eol eol eol E eol
```



Yacc (ix)

$L \rightarrow L\ E$ **eol** | L **eol** | ε
$E \rightarrow E$ **+** E | E **–** E | E ***** E | E **/** E | **(** E **)** | **–** E | **num**

```
%{
#include <stdio.h>
#include <ctype.h>
%}
%token NUM
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines   :  lines expr '\n' {printf("%d\n", $2);}
        |  lines '\n'
        |  /* ε */
        ;
expr    :  expr '+' expr   {$$ = $1 + $3;}
        |  expr '-' expr   {$$ = $1 - $3;}
        |  expr '*' expr   {$$ = $1 * $3;}
        |  expr '/' expr   {$$ = $1 / $3;}
        |  '(' expr ')'    {$$ = $2;}
        |  '-'  expr %prec UMINUS {$$ = -$2;}
        |  NUM
        ;
%%
```

```
yylex()
{
  int c;

  while((c = getchar()) == ' ' || c == '\t')
     ;
  if (isdigit(c))
  {
    ungetc(c, stdin);
    scanf("%d", &yylval);
    return(NUM);
  }
  return(c);
}

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

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

Yacc (x)

Options of Yacc:

`-d` (header) : generates `file.h` = declarations of exportable information

to make visible definition of tokens

```
#define YYSTYPE int

Symbols for Lex

extern YYSTYPE yylval;
```

`-v` (verbose) : generates `file.output` = description of LALR(1) parsing table

Pragmatically: **First**: run Yacc on G (without semantic actions, auxiliary functions, etc.) to be sure that the generated parser conforms to expectations.
Then: completion.

`YYDEBUG` : Tracing of the execution of the parser generated by Yacc (not tracing of Yacc!)

Symbol that must be defined, typically with **-D** option of C compiler: **-DYYDEBUG**

Actual tracing enabled by integer variable `yydebug`:

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

⇨ list of the actions executed by the parser for a given input

Yacc (xi)

- Generalization of type of values computed by semantic actions
  (in other words: type of pseudo-variables, e.g. calculator for real values)

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

- In general: different value types for different grammar symbols:

```
float  expr → expr addop term | term
float  term → term mulop factor | factor
float  factor → ( expr ) | rnum
 char  addop → + | –
 char  mulop → * | /
```

⇨ need to define `YYSTYPE` polymorphically = union { `float` `char`

⇩

2 ways: managed by { Yacc, user

Yacc (xii)

line → *expr* **eol**
expr → *expr addop term* | *term*
term → *term mulop factor* | *factor*
factor → **(** *expr* **)** | **rnum**
addop → **+** | **–**
mulop → ***** | **/**

1. By `%union`

```
...
%union {float value; char operator;}
%type <value> expr term factor RNUM
%type <operator> addop mulop
%%
line    :  expr '\n' {printf("%f\n", $1);}
        ;

expr    :  expr addop term
           {switch($2)
             {case '+': $$ = $1 + $3; break;
              case '-': $$ = $1 - $3; break;}}
        |  term
        ;

term    :  term mulop factor
           {switch($2)
             {case '*': $$ = $1 * $3; break;
              case '/': $$ = $1 / $3; break;}}
        |  factor
        ;
```

```
factor  :  '(' expr ')'  {$$ = $2;}
        |  RNUM
        ;

addop   :  '+'   {$$ = '+';}
        |  '-'   {$$ = '-';}
        ;

mulop   :  '*'   {$$ = '*';}
        |  '/'   {$$ = '/';}
        ;
```

Yacc (xiii)

2. By defining the data type in a separate file `typedef ... TYPE;`

```
#define YYSTYPE TYPE
```

(in file Yacc)

Value of `TYPE` constructed ad hoc within semantic actions

```
$$.field = $1.field1 + $2.field2
```

Example: Construction of the syntax tree: `typedef ... *PNODE;`

pointer to node of syntax tree

Yacc (xiv)

```
%{
#include <ctype.h>
#include <stdio.h>
typedef union {float value; char operator;} Value;
#define YYSTYPE Value
%}
%token RNUM
%%
line : expr '\n' {printf("%f\n", $1.value);}
     ;

expr : expr addop term
       {switch($2.operator)
         {case '+': $$.value = $1.value + $3.value; break;
          case '-': $$.value = $1.value - $3.value; break;}}
     | term
     ;

term : term mulop factor
       {switch($2.operator)
         {case '*': $$.value = $1.value * $3.value; break;
          case '/': $$.value = $1.value / $3.value; break;}}
     | factor
     ;
```

line → *expr* **eol**
expr → *expr addop term* | *term*
term → *term mulop factor* | *factor*
factor → **(** *expr* **)** | **rnum**
addop → **+** | **–**
mulop → ***** | **/**

```
factor : '(' expr ')'
         {$$.value = $2.value;}
       | RNUM
       ;

addop : '+' {$$.operator = '+';}
      | '-' {$$.operator = '-';}
      ;

mulop : '*'  {$$.operator = '*';}
      | '/'  {$$.operator = '/';}
      ;
```

Yacc (xv)

- Embedded semantic actions: when necessary executing code before the complete recognition of a production

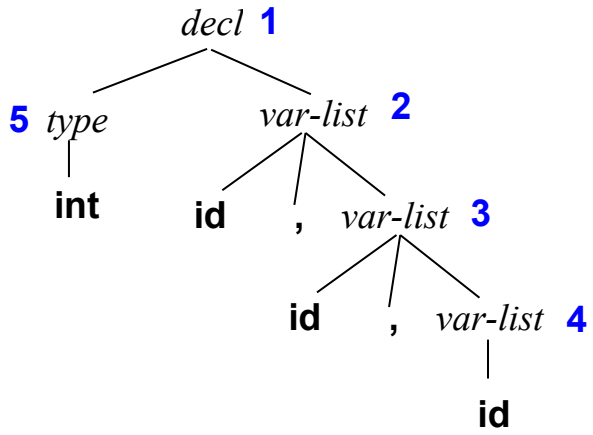
decl → *type var-list* **;**
type → **int** | **float**
var-list → **id ,** *var-list* | **id**

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

Goal: Analyzing identifiers in *var-list*, qualify each **id** with relevant type.

```
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);}
            ;
```

static variable



- Interpretation of embedded actions by Yacc:

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

≡

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

E → ε reduced *after action on* B

ε-*production*

Yacc (xvi)

- Internal identifiers of Yacc:

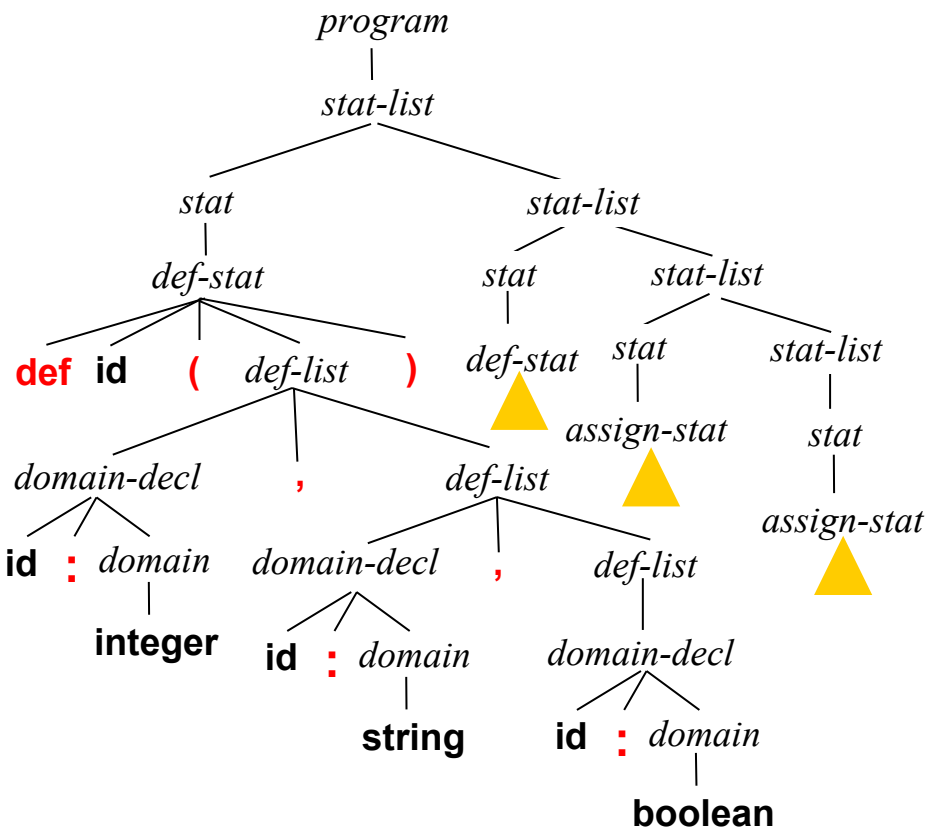
Identifier	*Description*
`file.c`	Name of output file
`file.h`	Header file generated by Yacc (using **-d**), containing `#define` of tokens
`yyparse()`	Parsing function
`yylval`	Value of current token (in stack)
`YYSTYPE`	Symbol for C preprocessor defining the type of values computed by semantic actions
`yydebug`	Integer variable enabling the execution of tracing of parser generated by Yacc

- Yacc definitions:

Keyword	*Definition*
`%token`	Symbol of preprocessing for tokens
`%start`	Axiom
`%union`	Union YYSTYPE to allow computing values of different types in semantic actions
`%type`	Variant type associated with a grammar symbol
`%left`	Left associativity for tokens
`%right`	Right associativity for tokens
`%nonassoc`	Non-associativity

Bottom-up Construction of (almost) Concrete Tree

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**




```
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*);

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

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

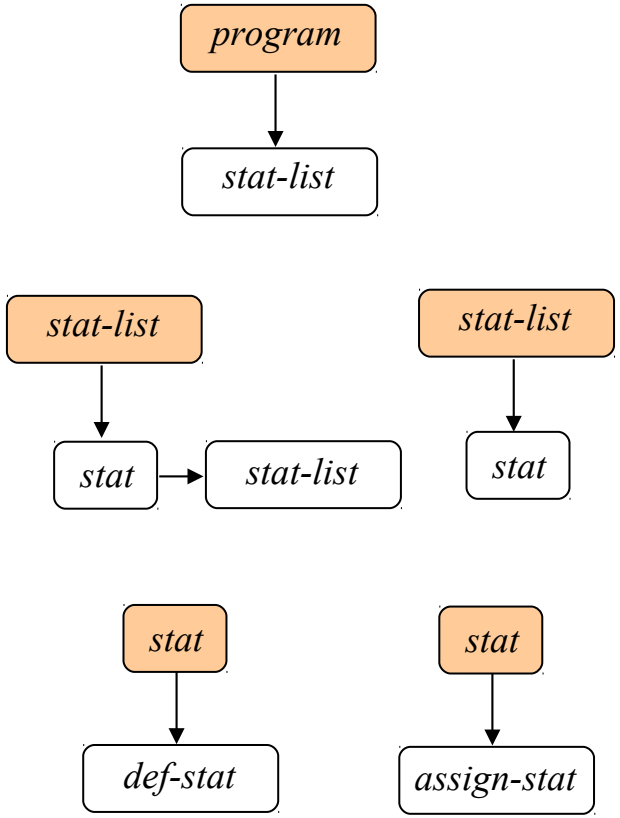
```
#define  DEF         258
#define  INTEGER     259
#define  STRING      260
#define  BOOLEAN     261
#define  ID          262
#define  INTCONST    263
#define  STRCONST    264
#define  BOOLCONST   265
#define  ASSIGN      266
#define  ERROR       267
```

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;}
     ;
```

lexical analyzer

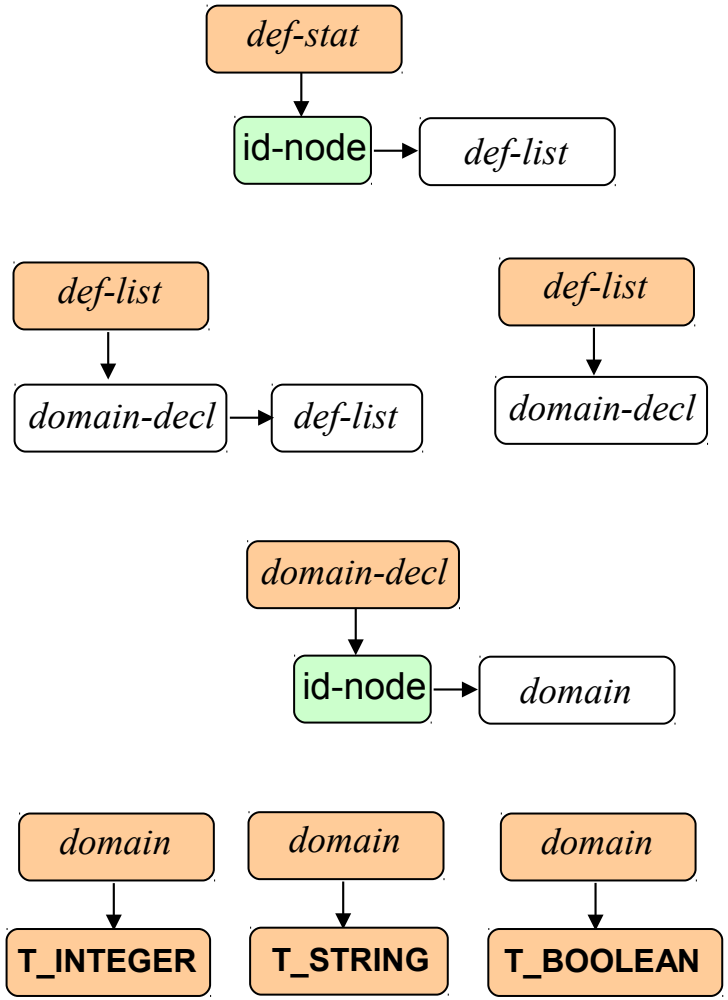
program → *stat-list*
stat-list → *stat stat-list* | *stat*
stat → *def-stat* | *assign-stat*



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**

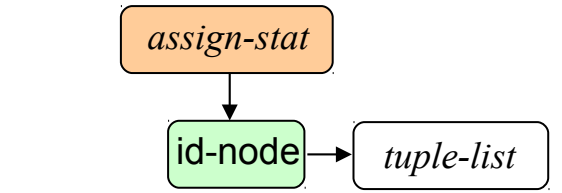
```
def_stat : DEF
           ID {$$ = idnode();}
           '(' def_list ')' {$$ = nontermnode(NDEF_STAT);
                             $$->child = $3;
                             $3->brother = $5;}
         ;

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);}
       ;
```

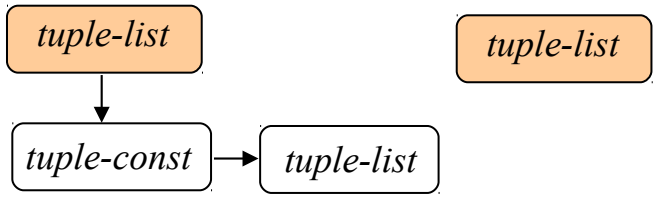


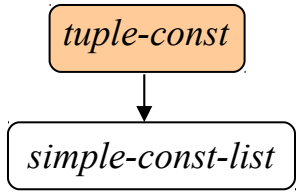
parser.y (iii)

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;}
            ;
```



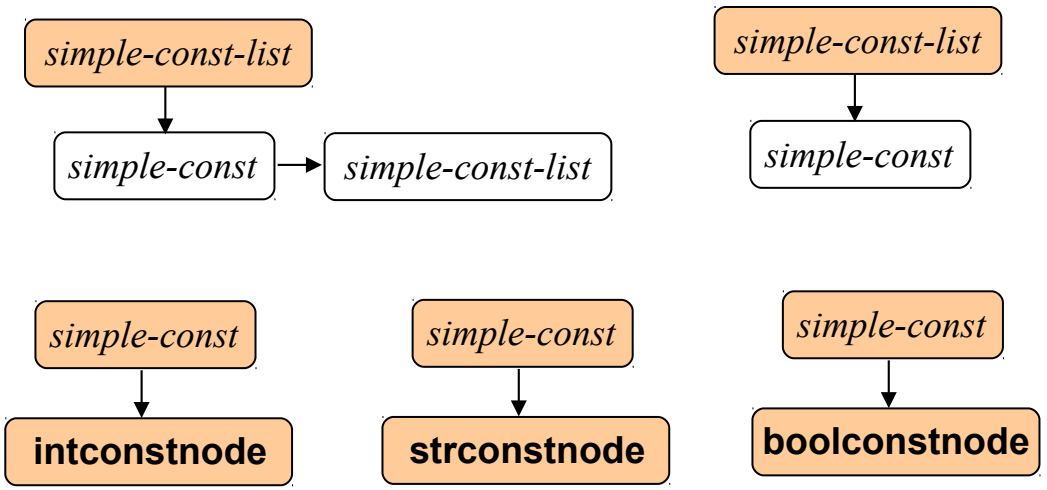




parser.y (iv)

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);
}

main()
{
  int result;

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

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

makefile

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

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

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

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

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

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

Bottom-up Construction of Abstract Tree (EBNF-like)

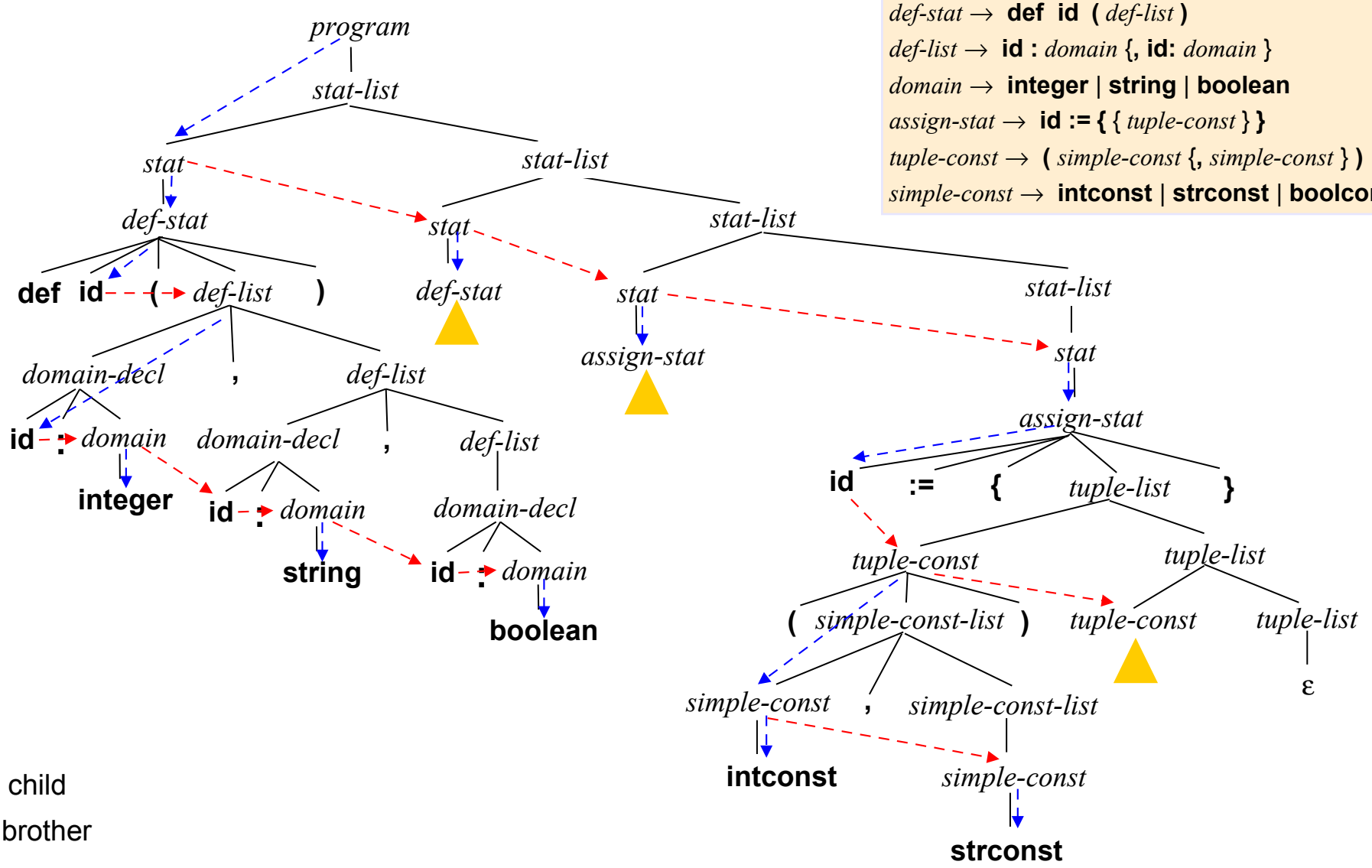
program → *stat-list*
stat-list → *stat stat-list* | *stat*
stat → *def-stat* | *assign-stat*
def-stat → **def id (** *def-list* **)**
def-list → *domain-decl* **,** *def-list* | *domain-decl*
domain-decl → **id :** *domain*
domain → **integer** | **string** | **boolean**
assign-stat → **id := {** *tuple-list* **}**
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**

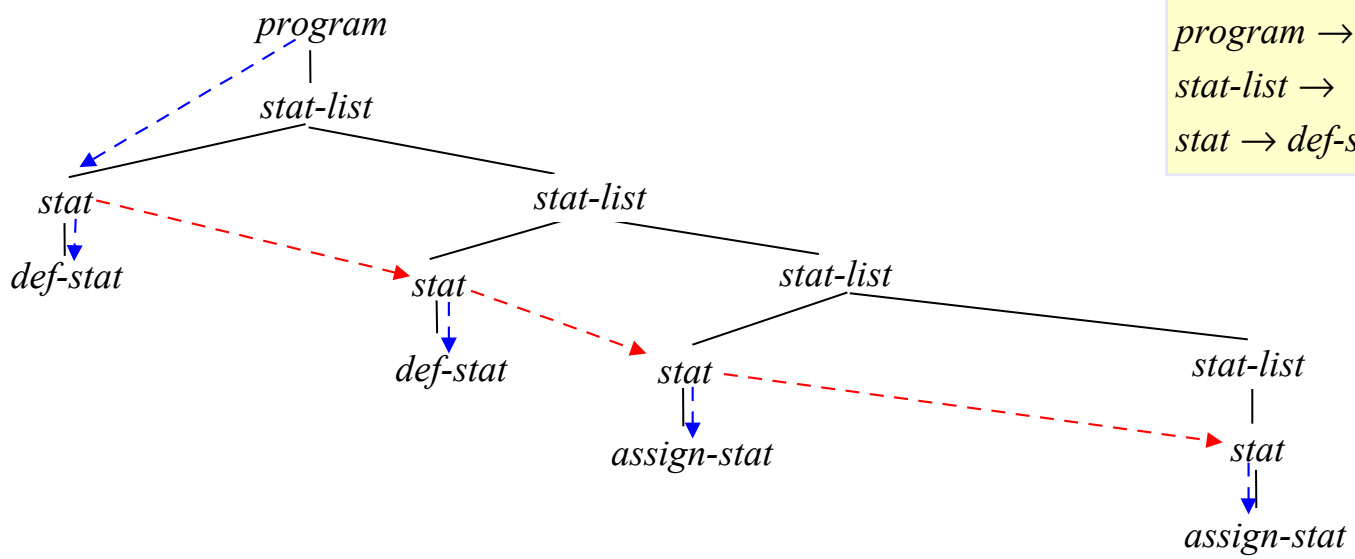
```
def R (A: integer, B: string, C: boolean)
def S (D: integer, E: string)
R := {(3, "alpha", true)(5, "beta", false)}
S := {(125, "sun")(236, "moon")}
```

Bottom-up Construction of Abstract Tree (EBNF-like) (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**



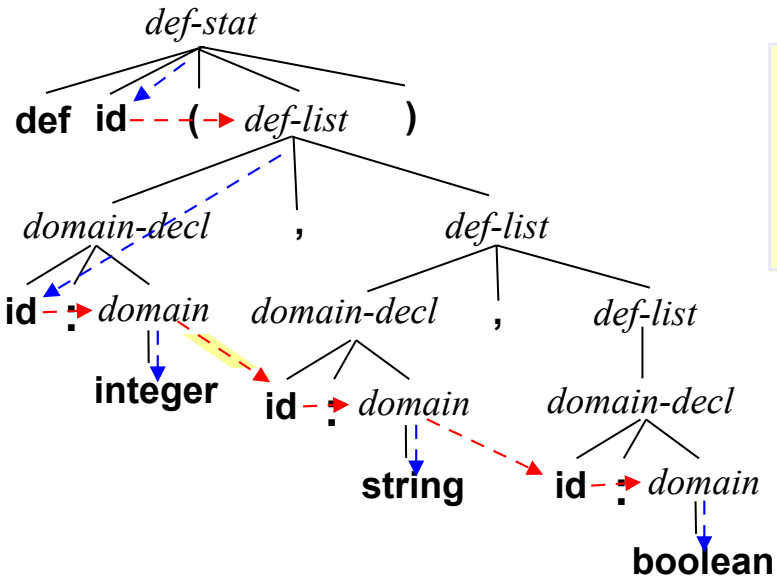
Bottom-up Construction of Abstract Tree (EBNF-like) (iii)



program → *stat-list*
stat-list → *stat stat-list* | *stat*
stat → *def-stat* | *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;}
     ;
```

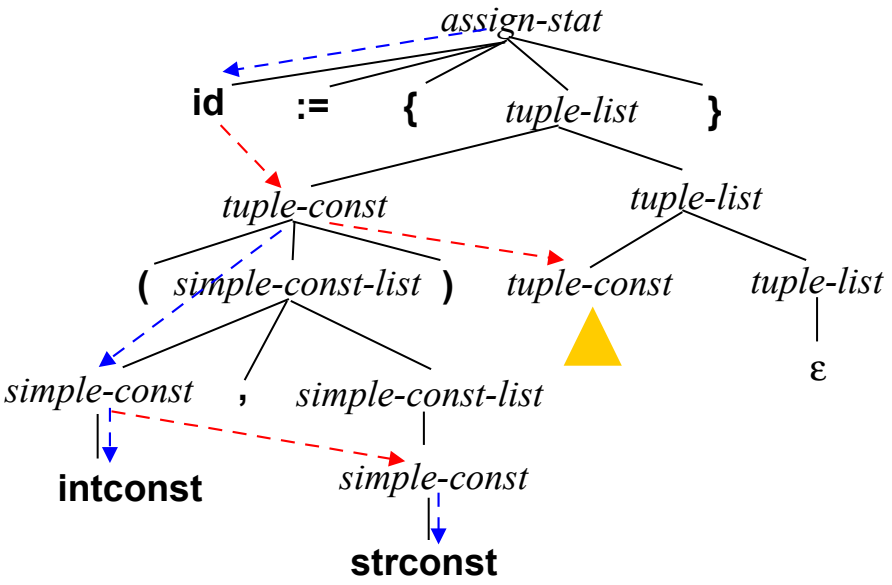
Bottom-up Construction of Abstract Tree (EBNF-like) (iv)



def-stat → **def id** (*def-list*)
def-list → *domain-decl* , *def-list* | *domain-decl*
domain-decl → **id** : *domain*
domain → **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);}
       ;
```

Bottom-up Construction of Abstract Tree (EBNF-like) (v)



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**

```
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();}
             ;
```