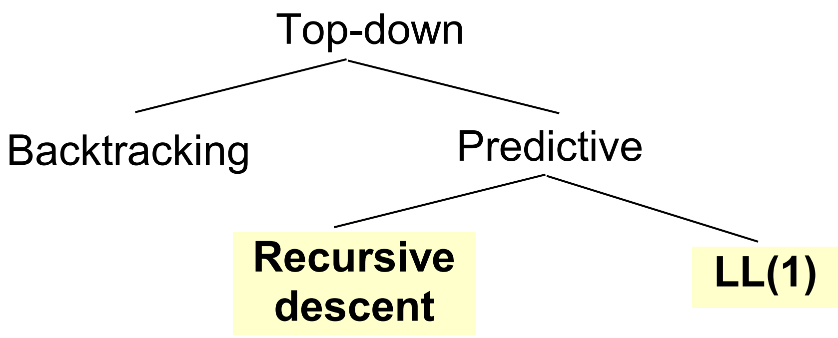
Top-down Parsing

- Corresponding to left canonical derivation



- Predictive: without backtracking → need to compute FIRST and FOLLOW sets ("signposting")

- **LL(1)**: in general **LL(K)**
  - **L** = left to right scanning
  - **L** = leftmost derivation
  - **K** = number of lookahead symbols necessary to decide

Top-down Parsing (ii)

type → *simple* **| ^ id | array [** *simple* **] of** *type*
simple → **integer** | **char** | **num dots num**

- Generation of the tree (≈ derivation): starting at the root = axiom, repetition of:

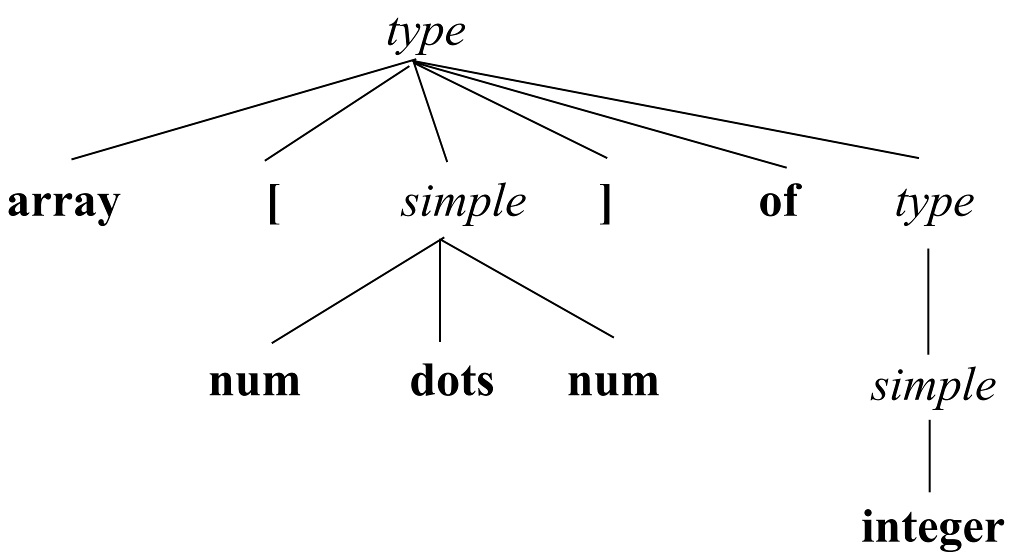
  1) At node *n* marked by A select a production of A and append to *n* its RHS;

  2) Choose next node (to expand). critical

- For some G: generation of the tree with a single scanning of the input

array [num dots num] of integer

lookahead (current symbol)



- In general → need for **backtracking**

Recursive-Descent Parsing

- Analysis of input by means of recursive procedures:

- $\forall$ nonterminal $\rightarrow$ association with a recursive (in general) procedure
  - Discrimination of alternatives (productions) based on lookahead symbol
  - $\forall$ alternative $\rightarrow$ scanning of RHS (list of grammar symbols):
    1. Terminal $\rightarrow$ match with lookahead symbol
    2. Nonterminal $\rightarrow$ call of corresponding procedure

- Structure of procedure associated with $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$:

```
procedure A()
begin
  if lookahead ∈ FIRST(α₁) then
    scan α₁
  else if lookahead ∈ FIRST(α₂) then
    scan α₂
  ...
  else if lookahead ∈ FIRST(αₙ) then
    scan αₙ
  else
    error()
end;
```

Set of lexical symbols starting instances of α_1

Omitted when $A \rightarrow \varepsilon$ is an alternative (in general, when: $\varepsilon \in \texttt{FIRST}(\alpha_i)$)

Recursive-Descent Parsing (ii)

type → *simple* | **^ id** | **array [** *simple* **] of** *type*
simple → **integer** | **char** | **num dots num**

```
var lookahead: Symbol;

procedure match(s: Symbol)
begin
  if lookahead = s then
    lookahead := next()
  else
    error()
end;
```

array [num dots num] of integer

lookahead

```
procedure type()
begin
  if lookahead in {integer, char, num} then
    simple()
  else if lookahead = '^' then
    begin
      match('^'); match(id)
    end
  else if lookahead = array then
    begin
      match(array); match('['); simple();
      match(']'); match(of); type()
    end
  else error()
end;
```

```
procedure simple()
begin
  if lookahead = integer then
    match(integer)
  else if lookahead = char then
    match(char)
  else if lookahead = num then
    begin
      match(num);
      match(dots);
      match(num)
    end
  else error()
end;
```

Recursive-Descent Parsing (iii)

- Extension to EBNF:

stat $\rightarrow$ **if** *expr* **then** *stat* [**else** *stat*]

```
procedure stat()
  match(IF); expr(); match(THEN); stat();
  if lookahead = ELSE then
    match(ELSE); stat()
  endif
end;
```

expr $\rightarrow$ *term* { **+** *term* }

```
procedure expr()
  term();
  while lookahead = PLUS do
    match(PLUS); term()
  endwhile
end;
```

stat-list $\rightarrow$ { *stat* **;** }$^{+}$
stat $\rightarrow$ **id :=** *expr*

```
procedure stat_list()
  do
    stat(); match(SEMICOLON)
  while lookahead = ID
end;
```

Recursive-Descent Parsing: Language for Tables

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")}
```

Recursive-Descent Parsing: Language for Tables (ii)

```
void parse()
{
  next();
  program();
}

void program()
{
  stat();
  while (lookahead == DEF || lookahead == ID)
    stat();
}

void stat()
{
  if (lookahead == DEF)
    def_stat();
  else if (lookahead == ID)
    assign_stat();
  else
    parserror();
}

void def_stat()
{
  match(DEF);
  match(ID);
  match('(');
  def_list();
  match(')');
}
```

program → *stat* { *stat* }

stat → *def-stat* | *assign-stat*

def-stat → **def** **id** **(** *def-list* **)**

def-list → **id** **:** *domain* {**, id:** *domain* }

```
void def_list()
{
  match(ID);
  match(':');
  domain();
  while(lookahead == ',')
  {
    next();
    match(ID);
    match(':');
    domain();
  }
}
```

Recursive-Descent Parsing: Language for Tables (iii)

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

assign-stat → **id := {** { *tuple-const* } **}**

tuple-const → **(** *simple-const* {**,** *simple-const* } **)**

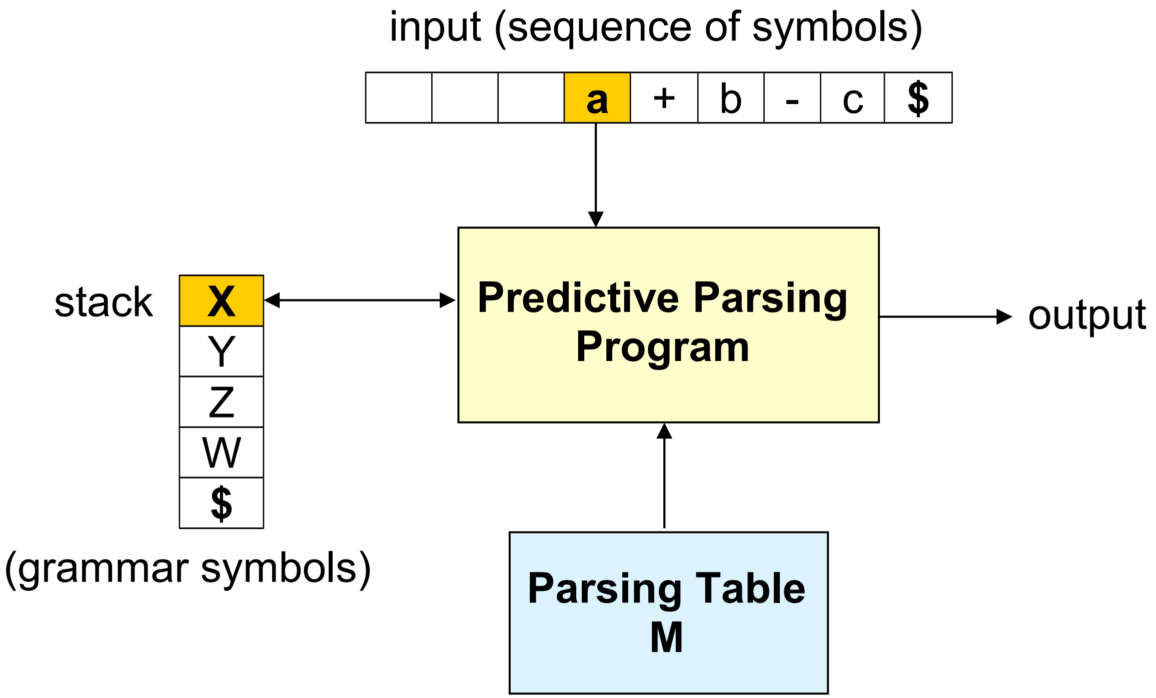
simple-const → **intconst** | **strconst** | **boolconst**

```
void domain()
{
  if (lookahead == INTEGER ||
      lookahead == STRING ||
      lookahead == BOOLEAN)
    next();
  else
    parserror();
}

void assign_stat()
{
  match(ID);
  match(ASSIGN);
  match('{');
  while ( lookahead == '(')
    tuple_const();
  match('}');
}

void tuple_const()
{
  match('(');
  simple_const();
  while ( lookahead == ',')
  {
    next();
    simple_const();
  }
  match(')');
}
```

```
void simple_const()
{
  if (lookahead == INTCONST ||
      lookahead == STRCONST ||
      lookahead == BOOLCONST)
    next();
  else
    parserror();
}

int main()
{
  parse();
  return(0);
}
```

LL(1) Parsing

- Direct management of a stack with support of a table for choosing the productions to expand



- Program (controller of the parsing process): based on (**X**, **a**) $\rightarrow$ action to execute:

1. X = a = **$**: accept
2. X = a $\neq$ **$**: pop(X); advance
3. Terminal(X), X $\neq$ a: error
4. Nonterminal(X): M[X,a] =
   - production of X, e.g: X $\rightarrow$ AbC $\Longrightarrow$ substitute X with CbA (top)
   - error (call to recovery routine)

Algorithm of LL(1) Parsing

- Input: w = string of terminals, M = parsing table for G
- Output: $w \in L(G) \rightarrow$ leftmost derivation for w, or error message

stack := $\begin{array}{|c|}\hline S \\ \hline \$ \\ \hline\end{array}$; input := w**$**; pc points to first symbol of w**$**;

repeat
    X := grammar symbol on top of the stack;
    a := terminal symbol pointed by pc;
    if Terminal(X) **or** X = **$ then**
        if X = a **then**
            pop(); pc++
        else error()
    else if M[X,a] = $X \rightarrow Y_1 Y_2 \ldots Y_n$ **then**
        pop();
        push($Y_n, Y_{n-1}, \ldots, Y_1$); /* Y_1 on top of the stack */
        print "$X \rightarrow Y_1 Y_2 \ldots Y_n$"
    else error()
until X = **$**.

LL(1) Parsing: Examples

$S \to (\, S \,) \, S \mid \varepsilon$ (strings of balanced parentheses)

M

	(	)	$
S	$S \to (\, S \,) \, S$	$S \to \varepsilon$	$S \to \varepsilon$

("expand" function)

w = ( )

Stack	*Input*	*Action*
$S$$	()$	$S \to (\, S \,) \, S$
(S)$S$$	()$	match
S)$S$$	)$	$S \to \varepsilon$
)$S$$	)$	match
$S$$	$	$S \to \varepsilon$
$	$	accept

LL(1) Parsing: Examples (ii)

$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (\ E\ ) \mid$ **num**

$\Rightarrow$

$E \to T\,E'$
$E' \to + T\,E' \mid \varepsilon$
$T \to F\,T'$
$T' \to * F\,T' \mid \varepsilon$
$F \to (\ E\ ) \mid$ **num**

	num	**+**	$*$	**(**	**)**	**$**
E	$E \to T\,E'$			$E \to T\,E'$		
E'		$E' \to + T\,E'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to F\,T'$			$T \to F\,T'$		
T'		$T' \to \varepsilon$	$T' \to * F\,T'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ **num**			$F \to (\ E\ )$		

w = **(num + num)** $*$ **num**

Stack	*Input*	*Action*
E$	(n+n)∗n$	$E \to TE'$
TE'$	(n+n)∗n$	$T \to FT'$
FT'E'$	(n+n)∗n$	$F \to (E)$
(E)T'E'$	(n+n)∗n$	match
E)T'E'$	n+n)∗n$	$E \to TE'$
TE')T'E'$	n+n)∗n$	$T \to FT'$
FT'E')T'E'$	n+n)∗n$	$F \to$ **num**
nT'E')T'E'$	n+n)∗n$	match
T'E')T'E'$	+n)∗n$	$T' \to \varepsilon$
E')T'E'$	+n)∗n$	$E' \to +TE'$
+TE')T'E'$	+n)∗n$	match
TE')T'E'$	n)∗n$	$T \to FT'$
FT'E')T'E'$	n)∗n$	$F \to$ **num**
nT'E')T'E'$	n)∗n$	match
T'E')T'E'$	)∗n$	$T' \to \varepsilon$
E')T'E'$	)∗n$	$E' \to \varepsilon$
)T'E'$	)∗n$	match
T'E'$	∗n$	$T' \to *FT'$
∗ FT'E'$	∗n$	match
FT'E'$	n$	$F \to$ **num**
nT'E'$	n$	match
T'E'$	$	$T' \to \varepsilon$
E'$	$	$E' \to \varepsilon$
$	$	accept

Notes:

- LL(1) ≈ recursive-descent parsing
- Parsing table: guides parsing
- Difference: generality of LL(1)
- Invariance of left-recursion problem

Instantiation of Parsing Table: FIRST

- Conceptually: $FIRST(\alpha) \supseteq \{ a \mid \alpha \overset{*}{\Rightarrow} a\beta \}$; if $\alpha \overset{*}{\Rightarrow} \varepsilon$, then $\varepsilon \in FIRST(\alpha)$.

- Inductively:

a) Given X = grammar symbol or ε, $FIRST(X)$ = { terminal [+ ε] } defined as follows:

1. If X is a terminal or ε, then $FIRST(X) = \{ X \}$

2. If X is a nonterminal, then $\forall$ alternative $X \to \alpha$ ($FIRST(X) \supseteq FIRST(\alpha)$)

b) Given $\alpha = X_1 X_2 ... X_n$ = string of grammar symbols, $FIRST(\alpha)$ is defined as follows:

- $FIRST(\alpha) \supseteq FIRST(X_1) - \{\varepsilon\}$
- If $\exists\, i<n$ ($\varepsilon \in FIRST(X_1)$, $\varepsilon \in FIRST(X_2)$, ..., $\varepsilon \in FIRST(X_i)$) then $FIRST(\alpha) \supseteq FISRT(X_{i+1}) - \{\varepsilon\}$
- If $\forall\, i \in [1..n]$ ($\varepsilon \in FIRST(X_i)$) then $\varepsilon \in FIRST(\alpha)$

FIRST: Examples

1.

$E \rightarrow T\, E'$
$E' \rightarrow \mathbf{+}\; T\, E' \mid \varepsilon$
$T \rightarrow F\, T'$
$T' \rightarrow \mathbf{*}\; F\, T' \mid \varepsilon$
$F \rightarrow \mathbf{(}\; E\; \mathbf{)} \mid \mathbf{id}$

$\Rightarrow$

$FIRST(E) = FIRST(T) = FIRST(F) = \{\ \mathbf{(},\ \mathbf{id}\ \}$
$FIRST(E') = \{\ \mathbf{+},\ \varepsilon\ \}$
$FIRST(T') = \{\ \mathbf{*},\ \varepsilon\ \}$

2.

$stat \rightarrow$ *if-stat* | **other**
if-stat $\rightarrow$ **if** *expr* **then** *stat else-part*
else-part $\rightarrow$ **else** *stat* | ε
expr $\rightarrow$ **true** | **false**

$\Rightarrow$

FIRST(*stat*) = { **if**, **other** }
FIRST(*if-stat*) = { **if** }
FIRST(*else-part*) = { **else**, ε }
FIRST(*expr*) = { **true**, **false** }

3.

stat-list $\rightarrow$ *stat stat-list'*
stat-list' $\rightarrow$ **;** *stat stat-list'* | ε
stat $\rightarrow$ **s**

$\Rightarrow$

FIRST(*stat-list*) = *FIRST*(*stat*) = { **s** }
FIRST(*stat-list'*) = { **;**, ε }

Instantiation of Parsing Table: FOLLOW

- Lookahead set: necessary in both LL(1) and recursive-descent parsing
- *FIRST* not enough $\Rightarrow$ necessary *FOLLOW* when $A \to \alpha$, $\alpha \overset{*}{\Rightarrow} \varepsilon$
- Conceptually: $FOLLOW(A) \supseteq \{\, a \mid S \overset{*}{\Rightarrow} \alpha A a \beta \,\}$

  Also: if $S \overset{*}{\Rightarrow} \alpha A$, then **$** $\in FOLLOW(A)$.

- Inductively: Given a nonterminal A, $FOLLOW(A)$ = { terminals [+ **$**] } defined by the following rules:
  - If A = axiom, then **$** $\in FOLLOW(A)$;
  - If $\exists$ production $B \to \alpha A \gamma$, then $FOLLOW(A) \supseteq FIRST(\gamma) - \{\, \varepsilon \,\}$;
  - If $\exists$ production $B \to \alpha A \gamma$ such that $\varepsilon \in FIRST(\gamma)$, then $FOLLOW(A) \supseteq FOLLOW(B)$.

FOLLOW: Examples

1.

$E \to T\,E'$
$E' \to \mathbf{+}\; T\,E' \mid \varepsilon$
$T \to F\,T'$
$T' \to *\; F\,T' \mid \varepsilon$
$F \to \mathbf{(}\; E\; \mathbf{)} \mid \mathbf{id}$

$FIRST(E') = \{ \mathbf{+}, \varepsilon \}$
$FIRST(T') = \{ *, \varepsilon \}$

⇨

$FOLLOW(E) = \{ \mathbf{)}, \mathbf{\$} \}$
$FOLLOW(E') = \{ \mathbf{)}, \mathbf{\$} \}$
$FOLLOW(T) = \{ \mathbf{+}, \mathbf{)}, \mathbf{\$} \}$
$FOLLOW(T') = \{ \mathbf{+}, \mathbf{)}, \mathbf{\$} \}$
$FOLLOW(F) = \{ *, \mathbf{+}, \mathbf{)}, \mathbf{\$} \}$

2.

stat → *if-stat* | **other**
if-stat → **if** *expr* **then** *stat else-part*
else-part → **else** *stat* | ε
expr → **true** | **false**

FIRST(*stat*) = { **if**, **other** }
FIRST(*if-stat*) = { **if** }
FIRST(*else-part*) = { **else**, ε }
FIRST(*expr*) = { **true**, **false** }

⇨

FOLLOW(*stat*) = { **$**, **else** }
FOLLOW(*if-stat*) = { **$**, **else** }
FOLLOW(*else-part*) = { **$**, **else** }
FOLLOW(*expr*) = { **then** }

3.

stat-list → *stat stat-list'*
stat-list' → **;** *stat stat-list'* | ε
stat → **s**

FIRST(*stat-list*) = *FIRST*(*stat*) = { **s** }
FIRST(*stat-list'*) = { **;**, ε }

⇨

FOLLOW(*stat-list*) = { **$** }
FOLLOW(*stat-list'*) = { **$** }
FOLLOW(*stat*) = { **;**, **$** }

Algorithm for Constructing Parsing Table M[A,a]

```
for each nonterminal A do
    for each alternative A → α do
        for each a ∈ FIRST(α), a ≠ ε do
            Insert A → α in M[A,a]
        end-for
        if ε ∈ FIRST(α) then
            for each a ∈ FOLLOW(A) do
                Insert A → α in M[A,a]
            end-for
        end-if
    end-for
end-for.
```

- <u>Def</u>: G is LL(1) if and only if parsing table M is not ambiguous.

Construction of Parsing Table M: Examples

1.
$E \rightarrow T E'$
$E' \rightarrow + T E' \mid \varepsilon$
$T \rightarrow F T'$
$T' \rightarrow * F T' \mid \varepsilon$
$F \rightarrow (E) \mid \mathbf{id}$

$\Rightarrow$

$FIRST(E) = FIRST(T) = FIRST(F) = \{ (, \mathbf{id} \}$
$FIRST(E') = \{ +, \varepsilon \}$
$FIRST(T') = \{ *, \varepsilon \}$

$FOLLOW(E) = FOLLOW(E') = \{), \$ \}$
$FOLLOW(T) = FOLLOW(T') = \{ +,), \$ \}$
$FOLLOW(F) = \{ *, +,), \$ \}$

$\Rightarrow$

$FIRST(TE') = \{ (, \mathbf{id} \}$
$FIRST(+TE') = \{ + \}$
$FIRST(FT') = \{ (, \mathbf{id} \}$
$FIRST(*FT') = \{ * \}$
$FIRST((E)) = \{ (\}$
$FIRST(\mathbf{id}) = \{ \mathbf{id} \}$

	id	+	*	(	)	$
E	$E \rightarrow T E'$			$E \rightarrow T E'$		
E'		$E' \rightarrow + T E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F T'$			$T \rightarrow F T'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow * F T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

- Since M is not ambiguous, G is LL(1)

Construction of Parsing Table M: Examples (ii)

2.

stat → *if-stat* | **other**
if-stat → **if** *expr* **then** *stat else-part*
else-part → **else** *stat* | ε
expr → **true** | **false**

⇨

FIRST(*stat*) = { **if**, **other** }
FIRST(*if-stat*) = { **if** }
FIRST(*else-part*) = { **else**, ε }
FIRST(*expr*) = { **true**, **false** }
FIRST(**other**) = { **other** }

FOLLOW(*stat*) = { **$**, **else** }
FOLLOW(*if-stat*) = { **$**, **else** }
FOLLOW(*else-part*) = { **$**, **else** }
FOLLOW(*expr*) = { **then** }

	if	**then**	**else**	**other**	**true**	**false**	**$**
stat	*stat* → *if-stat*			*stat* → **other**			
if-stat	*if-stat* → **if** *expr* **then** *stat else-part*						
else-part			*else-part* → **else** *stat* *else-part* → ε				*else-part* → ε
expr					*expr* → **true**	*expr* → **false**	

- Since M is ambiguous, G is <u>not</u> LL(1)

Construction of Parsing Table M: Examples (iii)

3.

$stat\text{-}list \to stat\ stat\text{-}list'$
$stat\text{-}list' \to \mathbf{;}\ stat\ stat\text{-}list' \mid \varepsilon$
$stat \to \mathbf{s}$

⇨

$FIRST(stat\text{-}list) = FIRST(stat) = \{\ \mathbf{s}\ \}$
$FIRST(stat\text{-}list') = \{\ \mathbf{;},\ \varepsilon\ \}$
$FIRST(stat\ stat\text{-}list') = \{\ \mathbf{s}\ \}$
$FIRST(\mathbf{;}\ stat\ stat\text{-}list') = \{\ \mathbf{;}\ \}$

$FOLLOW(stat\text{-}list) = FOLLOW(stat\text{-}list') = \{\ \mathbf{\$}\ \}$
$FOLLOW(stat) = \{\ \mathbf{;},\ \mathbf{\$}\ \}$

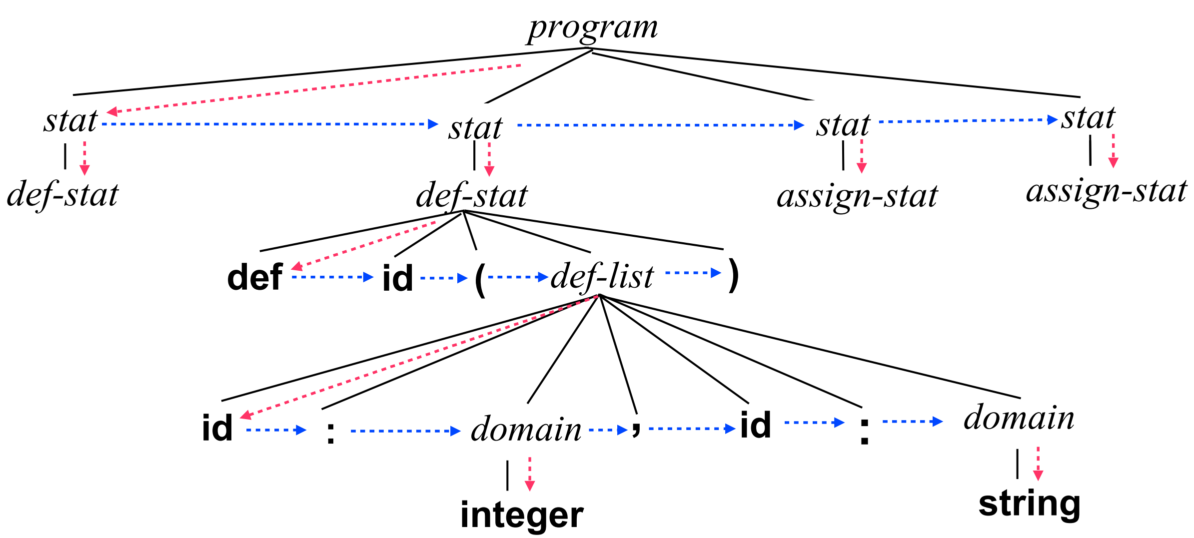
	;	**s**	**$**
stat-list		$stat\text{-}list \to stat\ stat\text{-}list'$	
stat-list'	$stat\text{-}list' \to \mathbf{;}\ stat\ stat\text{-}list'$		$stat\text{-}list' \to \varepsilon$
stat		$stat \to \mathbf{s}$	

- Since M is not ambiguous, G is LL(1)

Top-down Construction of Abstract Syntax Tree

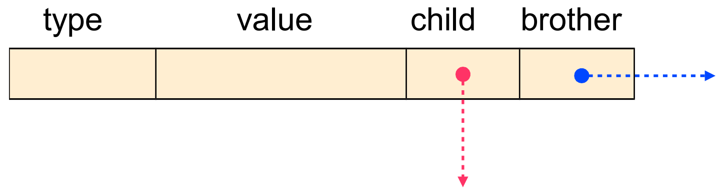
program → *stat* { *stat* }
stat → *def-stat* | *assign-stat*
def-stat → **def id (** *def-list* **)**
def-list → **id :** *domain* **{, id:** *domain* }
domain → **integer** | **string** | **boolean**
assign-stat → **id := {** { *tuple-const* } **}**
tuple-const → **(** *simple-const* **{,** *simple-const*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



Top-down Construction of Abstract Syntax Tree (ii)

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

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



- Rules of node “state”:

1. keyword / special character / operator ⇨ `type` = identifying constant (`DEF`, `LEFT`, `ASSIGN`, ...)

   syntax sugar: may be missing in the tree

2. integer constant ⇨ `type = T_INTCONST`, `value.ival`

   string constant ⇨ `type = T_STRCONST`, `value.sval`

   boolean constant ⇨ `type = T_BOOLCONST`, `value.bval`

3. Identifier ⇨ `type = T_ID`, `value.sval`

4. Nonterminal ⇨ `type = T_NONTERMINAL`, `value.ival`

def.h

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

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


typedef enum
{
  NPROGRAM,
  NSTAT,
  NDEF_STAT,
  NDEF_LIST,
  NDOMAIN,
  NASSIGN_STAT,
  NTUPLE_CONST,
  NSIMPLE_CONST
} Nonterminal;
```

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

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

typedef Node *Pnode;
```

```c
void match(int),
     next(),
     parserror(),
     treeprint(Pnode, int);

char *newstring(char*);

Pnode nontermnode(Nonterminal),
      idnode(),
      keynode(Typenode),
      intconstnode(),
      strconstnode(),
      boolconstnode(),
      newnode(Typenode),
      program(),
      stat(),
      def_stat(),
      def_list(),
      domain(),
      assign_stat(),
      tuple_const(),
      simple_const();
```

lexer.lex

```
%{
#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);
}
```

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**

scanner.c

```
#include "def.h"

#define NUM_KEYWORDS 6
#define MAXIDENT 100

FILE *yyin;
int line = 1;
char *yytext = NULL;
Value lexval;
int i, k;

struct {char* name; int keyword;}
keywords[NUM_KEYWORDS] =
{
  "def", DEF,
  "integer", INTEGER,
  "string", STRING,
  "boolean", BOOLEAN,
  "false", BOOLCONST,
  "true", BOOLCONST
};
```

```
int yylex()
{ int cc, keyword;

  if(yytext == NULL) yytext = malloc(MAXIDENT+1);

  do
  { cc = fgetc(yyin);
    if(cc == '\n')
      line++;
  } while(cc == ' ' || cc == '\t' || cc == '\n');
  if(cc == '(' || cc == ')' || cc == '{' || cc == '}' || cc == ',')
    return(cc);
  else if(cc == ':')
  { if((cc = fgetc(yyin)) == '=')
      return(ASSIGN);
    else
    { ungetc(cc, yyin);
      return(':');
    }
  }
  else if(isalpha(cc))
  { i = 0;
    yytext[i++] = cc;
    while(isalnum(cc = fgetc(yyin)))
      yytext[i++] = cc;
    ungetc(cc, yyin);
    yytext[i] = '\0';
    if(keyword = lookup(yytext))
    { if(keyword == BOOLCONST)
        lexval.bval = (yytext[0] == 'f' ? FALSE : TRUE);
      return (keyword);
    }
    else
    { lexval.sval = newstring(yytext);
      return(ID);
    }
  }
...
```

white space

id, keyword, boolconst

scanner.c (ii)

```
...
else if(isdigit(cc))
  {
    i = 0;
    yytext[i++] = cc;
    while(isdigit(cc = fgetc(yyin)))
      yytext[i++] = cc;
    ungetc(cc, yyin);
    yytext[i] = '\0';
    lexval.ival = atoi(yytext);
    return(INTCONST);
  }
  else if(cc == '"')
  {
    i = 0;
    yytext[i++] = cc;
    while((cc = fgetc(yyin)) != '"')
      yytext[i++] = cc;
    yytext[i++] = cc;
    yytext[i] = '\0';
    lexval.sval = newstring(yytext);
    return(STRCONST);
  }
  else if(cc==EOF)
    return(EOF);
  else
    return(ERROR);
}
```

```
int lookup(char *id)
{
  for(k = 0; k < NUM_KEYWORDS; k++)
    if(strcmp(id, keywords[k].name) == 0)
      return(keywords[k].keyword);
  return(0);
}

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

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

parser.c

```
include "def.h"

extern char *yytext;
extern Value lexval;
extern int line;
extern FILE *yyin;

int lookahead;

Pnode root = NULL;

void next()
{
  lookahead = yylex();
}

void match(int symbol)
{
  if(lookahead == symbol)
    next();
  else
    parserror();
}

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

parser.c (ii)

```c
Pnode newnode(Typenode tnode)
{
  Pnode p;

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

Pnode nontermnode(Nonterminal nonterm)
{
  Pnode p;

  p = newnode(T_NONTERMINAL);
  p->value.ival = nonterm;
  return(p);
}

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

Pnode idnode()
{
  Pnode p;

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

```c
Pnode intconstnode()
{
  Pnode p;

  p = newnode(T_INTCONST);
  p->value.ival = lexval.ival;
  return(p);
}

Pnode strconstnode()
{
  Pnode p;

  p = newnode(T_STRCONST);
  p->value.sval = lexval.sval;
  return(p);
}

Pnode boolconstnode()
{
  Pnode p;

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

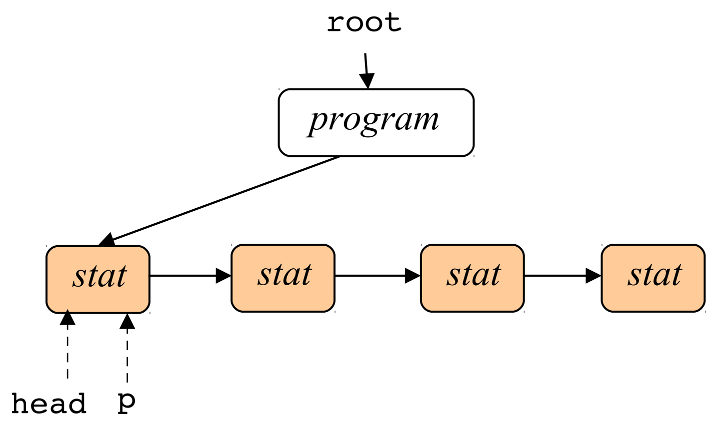
```c
int main()
{
  yyin = stdin;
  parse();
  treeprint(root, 0);
  return(0);
}
```

parser.c (iii)

```
void parse()
{
  next();
  root = nontermnode(NPROGRAM);
  root->child = program();
}
```

```
Pnode program()
{
  Pnode head, p;

  head = p = nontermnode(NSTAT);
  p->child = stat();
  while (lookahead == DEF || lookahead == ID)
  {
    p->brother = nontermnode(NSTAT);
    p = p->brother;
    p->child = stat();
  }
  return(head);
}
```

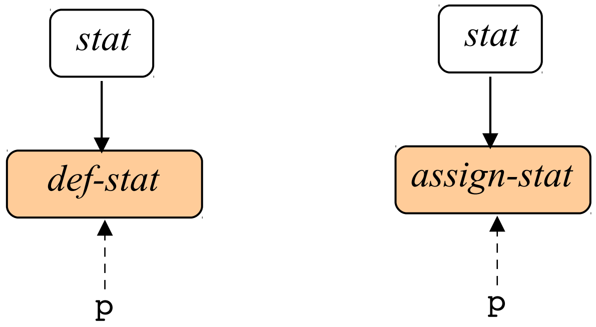
program $\rightarrow$ *stat* { *stat* }



parser.c (iv)

```
Pnode stat()
{
  Pnode p;

  if (lookahead == DEF)
  {
    p = nontermnode(NDEF_STAT);
    p->child = def_stat();
    return(p);
  }
  else if (lookahead == ID)
  {
    p = nontermnode(NASSIGN_STAT);
    p->child = assign_stat();
    return(p);
  }
  else
    parserror();
}
```

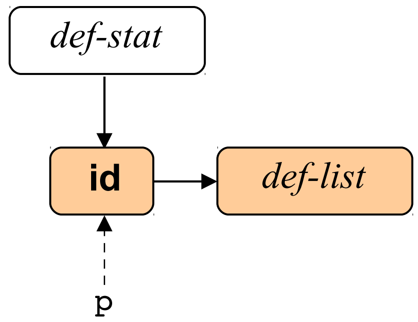
stat $\rightarrow$ *def-stat* | *assign-stat*



parser.c (v)

```
Pnode def_stat()
{
  Pnode p;

  match(DEF);
  if (lookahead == ID)
  {
    p = idnode();
    next();
    match('(');
    p->brother = nontermnode(NDEF_LIST);
    p->brother->child = def_list();
    match(')');
    return(p);
  }
  else
    parserror();
}
```

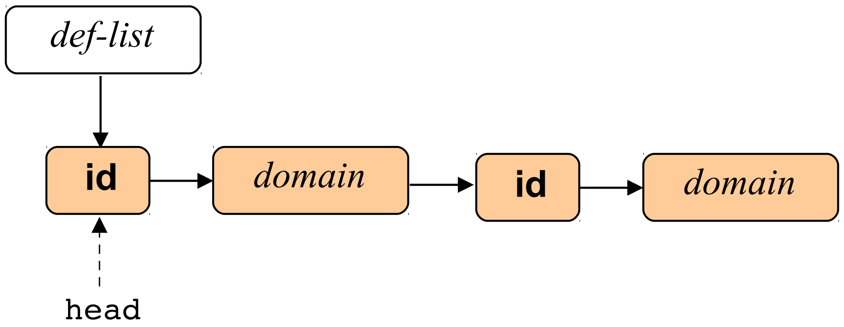
def-stat $\rightarrow$ **def id** (*def-list*)



parser.c (vi)

```
Pnode def_list()
{
  Pnode head, p;

  if (lookahead == ID)
  {
    head = p = idnode();
    next();
    match(':');
    p->brother = nontermnode(NDOMAIN);
    p = p->brother;
    p->child = domain();
    while(lookahead == ',')
    {
      next();
      if ( lookahead == ID)
      {
        p->brother = idnode();
        p = p->brother;
        next();
        match(':');
        p-> brother = nontermnode(NDOMAIN);
        p = p->brother;
        p->child = domain();
      }
      else
        parserror();
    }
    return(head);
  }
  else
    parserror();
}
```

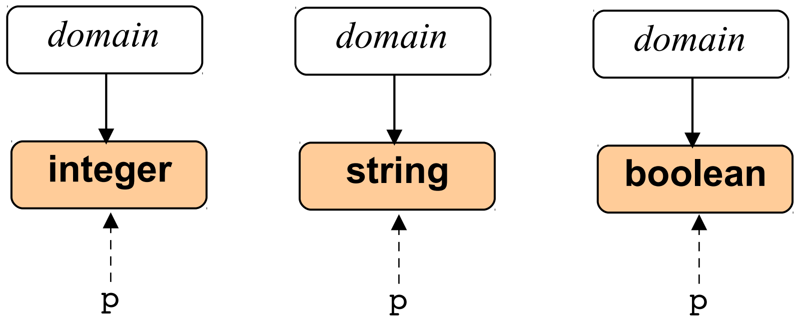
def-list $\rightarrow$ **id** : *domain* { , **id** : *domain* }



parser.c (vii)

```
Pnode domain()
{
  Pnode p;

  if (lookahead == INTEGER ||
      lookahead == STRING ||
      lookahead == BOOLEAN)
  {
    p = keynode( lookahead == INTEGER ? T_INTEGER :
                   ( lookahead == STRING ? T_STRING :
                        T_BOOLEAN));
    next();
    return(p);
  }
  else
    parserror();
}
```

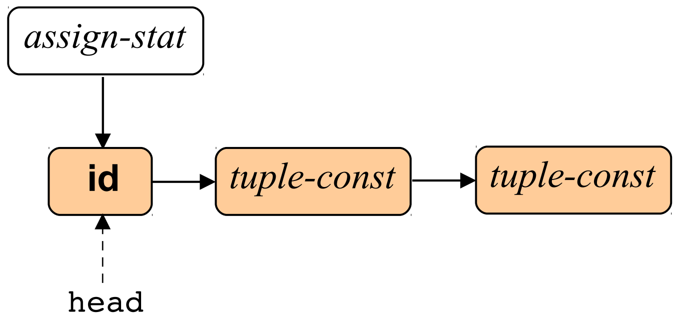
domain $\rightarrow$ **integer** | **string** | **boolean**



parser.c (viii)

```
Pnode assign_stat()
{
  Pnode head, p;

  if (lookahead == ID)
  {
    head = p = idnode();
    next();
    match(ASSIGN);
    match('{');
    while ( lookahead == '(')
    {
      p->brother = nontermnode(NTUPLE_CONST);
      p = p->brother;
      p->child = tuple_const();
    }
    match('}');
  }
  else
    parserror();
  return(head);
}
```

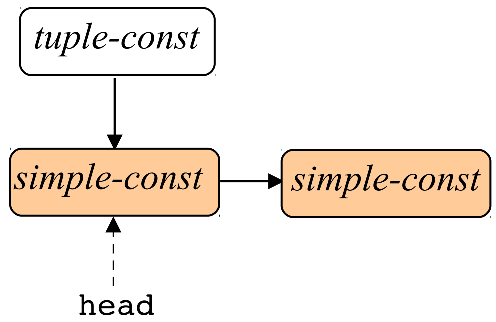
assign-stat → **id** := { { *tuple-const* } }



parser.c (ix)

```
Pnode tuple_const()
{
  Pnode head, p;

  match('(');
  head = p = nontermnode(NSIMPLE_CONST);
  p->child = simple_const();
  while (lookahead == ',')
  {
    next();
    p->brother = nontermnode(NSIMPLE_CONST);
    p = p->brother;
    p->child = simple_const();
  }
  match(')');
  return(head);
}
```

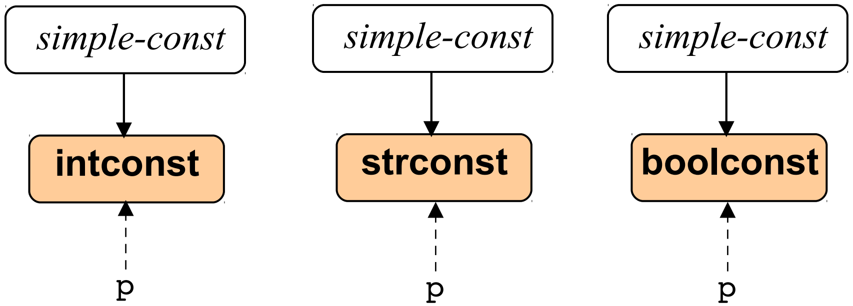
tuple-const $\rightarrow$ (*simple-const* { , *simple-const* })



parser.c (x)

```
Pnode simple_const()
{
  Pnode p;

  if (lookahead == INTCONST)
  {
    p = intconstnode();
    next();
    return(p);
  }
  else if (lookahead == STRCONST)
  {
    p = strconstnode();
    next();
    return(p);
  }
  else if (lookahead == BOOLCONST)
  {
    p = boolconstnode();
    next();
    return(p);
  }
  else
    parserror();
}
```

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



tree.c

```c
#include "def.h"

char* tabtypes[] =
{
"INTEGER",
"STRING",
"BOOLEAN",
"INTCONST",
"BOOLCONST",
"STRCONST",
"ID",
"NONTERMINAL"
};

char* tabnonterm[] =
{
  "PROGRAM",
  "STAT",
  "DEF_STAT",
  "DEF_LIST",
  "DOMAIN",
  "ASSIGN_STAT",
  "TUPLE_CONST",
  "SIMPLE_CONST"
};
```

```c
void treeprint(Pnode root, int indent)
{
  int i;
  Pnode p;

  for(i=0; i<indent; i++)
    printf("    ");
  printf("%s", (root->type == T_NONTERMINAL ? tabnonterm[root->value.ival] :
                                              tabtypes[root->type]));
  if(root->type == T_ID || root->type == T_STRCONST)
    printf(" (%s)", root->value.sval);
  else if(root->type == T_INTCONST)
    printf(" (%d)", root->value.ival);
  else if(root->type == T_BOOLCONST)
    printf(" (%s)", (root->value.ival == TRUE ? "true" : "false"));
  printf("\n");
  for(p=root->child; p != NULL; p = p->brother)
    treeprint(p, indent+1);
}
```

lexical value

makefile

```
syntax: syntax.o
	cc -g -o syntax syntax.o

tds: scanner.o parser.o tree.o
	cc -g -o tds scanner.o parser.o tree.o

tdl: lexer.o parser.o tree.o
	cc -g -o tdl lexer.o parser.o tree.o

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

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

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

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

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

lexer.c: lexer.lex def.h
	flex -o lexer.c lexer.lex
```

Execution

```
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")}
```

```
PROGRAM
    STAT
        DEF_STAT
            ID (R)
            DEF_LIST
                ID (A)
                DOMAIN
                    INTEGER
                ID (B)
                DOMAIN
                    STRING
                ID (C)
                DOMAIN
                    BOOLEAN
    STAT
        DEF_STAT
            ID (S)
            DEF_LIST
                ID (D)
                DOMAIN
                    INTEGER
                ID (E)
                DOMAIN
                    STRING
    STAT
        ASSIGN_STAT
            ID (R)
            TUPLE_CONST
                SIMPLE_CONST
                    INTCONST (3)
                SIMPLE_CONST
                    STRCONST ("alpha")
                SIMPLE_CONST
                    BOOLCONST (true)
            TUPLE_CONST
                SIMPLE_CONST
                    INTCONST (5)
                SIMPLE_CONST
                    STRCONST ("beta")
                SIMPLE_CONST
                    BOOLCONST (false)
    STAT
        ASSIGN_STAT
            ID (S)
            TUPLE_CONST
                SIMPLE_CONST
                    INTCONST (125)
                SIMPLE_CONST
                    STRCONST ("sun")
            TUPLE_CONST
                SIMPLE_CONST
                    INTCONST (236)
                SIMPLE_CONST
                    STRCONST ("moon")
```

Execution (ii)

```
def People (name: string, surname: string, age: integer)
People := {("Ann", "White", 24)("Louis", "Red", 40)("Lisa", "Green", 56)}
```

```
PROGRAM
    STAT
        DEF_STAT
            ID (People)
            DEF_LIST
                ID (name)
                DOMAIN
                    STRING
                ID (surname)
                DOMAIN
                    STRING
                ID (age)
                DOMAIN
                    INTEGER
    STAT
        ASSIGN_STAT
            ID (People)
            TUPLE_CONST
                SIMPLE_CONST
                    STRCONST ("Ann")
                SIMPLE_CONST
                    STRCONST ("White")
                SIMPLE_CONST
                    INTCONST (24)
            TUPLE_CONST
                SIMPLE_CONST
                    STRCONST ("Louis")
                SIMPLE_CONST
                    STRCONST ("Red")
                SIMPLE_CONST
                    INTCONST (40)
            TUPLE_CONST
                SIMPLE_CONST
                    STRCONST ("Lisa")
                SIMPLE_CONST
                    STRCONST ("Green")
                SIMPLE_CONST
                    INTCONST (56)
```