Flex , **Bison** and the ACSE compiler suite

Marcello M. Bersani

LFC – Politecnico di Milano

Bison example

- Calculator of parameterized formulae
- Simple compiler

Ex 1 – to do

- Implement a calculator handling formulae

2*(1+([x=3] (x+1)*([y=?] y+x)))=

- A formula is
  - an expression

(1+2*3)

Ex 1 – to do

- an expression with environment

  ([x=3] 1+x)

  - [x=n] defines an environment for variable x
- an expression with unknown environment

  ([x=?] 1+x)

  - [x=?] as above but the value is defined on-the-fly by the context (user)
    - Read x from console

- If an environment already exists for x, new nested definitions are rejected

Ex 1 - analysis

- Formulae are:
  - numbers or vars:

$$\text{expr} \rightarrow \text{NUM} \mid \text{VAR}$$

  - (expr)

$$\text{expr} \rightarrow \text{'(' expr ')'}$$

  - operations

$$\text{expr} \rightarrow \text{expr OP expr}$$

  - ([e] expr)

$$\text{expr} \rightarrow \text{' (' env expr ')'}$$

Ex 1 - analysis

- Environment [e] in ([e] expr)

$$\text{env} \rightarrow \text{VAR '=' v\_def}$$

$$\text{v\_def} \rightarrow \text{NUM | UNK}$$

Ex 1 – overview of grammar

- expr $\rightarrow$ NUM
  | VAR
  | expr PLUS expr
  | …
  | '(' env expr ')'

- env $\rightarrow$ '[' VAR '=' v_def ']'
- v_def $\rightarrow$ NUM | UNK

Ex 1 – overview of types

- **expr** → **NUM**
  | **VAR**
  | **expr** PLUS **expr**
  | …
  | '(' env **expr** ')'

```
%union{
  char c_var;
  int i_value;
  …
}
```

- env → '[' VAR '=' v_def ']'
- v_def → NUM | UNK

Ex 1 - environments

- Assume a finite number of [e]
- Use a stack **v** to represent nested [e]

  …([x=3] (x+1)*([y=1] …))

  – When parsing [e] push value of ‘var’ in **v**
  – Any occurrence of ‘var’ in **expr**

  ([e] **expr**)

  is replaced by the value in [e]
- When a formula ([e] expr) ends
  – Pop [e]

Ex 1 - environments

- Type of stack elements

```
struct VFREE{
   char var;
   int value;
};
```

- Stack (within the parser)

```
VFREE v[N];
int top;
```

Ex 1 - environments

- Getting value of a environment var

```
expr → VAR {
        int j, found=0;
        for(j=top-1; (j>=0 && !found); j--)
            if (v[j].var == $1){
                $$ = v[j].value;
                found = 1;
            }
        if (j<0 && !found) YYABORT;
    }
```

Ex 1 – environment formulae

- Getting value of an expr

```
expr → '(' expr ')'          { $$ = $2; }
      | '(' env expr ')'     {
                                 $$ = $3;
                                // pop env
                             }
```

only if the env was pushed on the stack

Ex 1 – managing stack

- First solution
  - the action for **env** manages the push operation
  - The semantic value of **env** is a boolean
    - True: a new env [e] is pushed
    - False: no new env [e]

  used to rule the pop

```
expr → '(' env expr ')'     {
                               $$ = $3;
                               if ($2) pop env
                            }
```

Ex 1 – managing stack

- Second solution
  - The semantic value of **env** is a struct ⟨var,value⟩
  - Push operation is done by a mid-action rule; its semantic value is a boolean
    - True: a new env [e] is pushed
    - False: no new env [e]

    used to rule the pop

Ex 1 – type

- Second solution

```
%union{
  char c_var;
  int i_value;
  struct {
        char var;
        int value;} env_def;
}
```

expr → '(' **env** { ... }
expr ')' { ... }

Ex 1 – managing stack

- Second solution

```
expr → '(' env      {
                        if ($2.var ∉ v)){
                          v.push($2)
                          $<i_value>$ = 1;}
                        else $<i_value>$ = 0;
                    }
          expr ')'  {
                        $$ = $4;
                        if ($<i_value>3) v.pop();
                    }
```

Ex 1 - environments

- Getting environment [e]

```
env → '[' VAR '=' v_def ']'  {
        int val;
        $$.var = $2;
        if ($4.var == '?'){
          printf("[ %c ]> ", $2);
          scanf("%d", &val);
          $$.value = val;
        }
        else
          $$.value = $4.value;
}
```

```
%union{
  char c_var;
  int i_value;
  struct {
        char var;
        int value;} env_def;
}
```

Ex 1 - environments

- Getting variable definition

env → '[' VAR '=' **v_def** ']' {

```
struct {
    char var;
    int value;
} env_def;
```

```
  …
  if ($4.var == '?') {…}
  else
      $$.value = $4.value;
}
```

```
v_def: NUMBER      { $$.value = $1; }
     | UNK         { $$.var = '?';  }
```

Ex 1 – managing stack

```
expr → '(' env   {
                   int j;

                   for(j=top-1; (j>=0 && v[j].var!=$2.var); j--);
                   if (j<0){
                           v[top].var = $2.var;
                           v[top].value = $2.value;
                           $<i_value>$ = 1;
                           top++;
                   }
                   else
                           $<i_value>$ = 0;
                 }
        expr ')' {   $$ = $4;
                     if ($<i_value>3) top--; };
```

Ex 1 - types

%union{
    char **c_var**;
    int **i_value**;
    struct {char var; int value;} **env_def**;
}

%token UNK
%token <**i_value**> NUMBER
%token <**c_var**> VAR
%type <**i_value**> expr
%type <**env_def**> env
%type <**env_def**> v_def

Ex 1 - lexer

```
%{
#include <stdlib.h>
#include "calc2gr.tab.h"
%}

%option noyywrap


%%

%%
```

```
[ \t]+  {}

"+" {return PLUS;}
"-" {return MINUS;}
"*" {return PER;}
"(" {return '(';}
")" {return ')';}
"[" {return '[';}
"]" {return ']';}
"=" {return '='; }
"?" {return UNKNOWN;}
[a-z] {yylval.c_var = yytext[0];
      return VAR;}

"-"?([0-9]|([1-9][0-9]*)) {
  yylval.i_value = atoi(yytext);
  return NUMBER;
}
```

```
%union{
  char c_var;
  int i_value;
  struct {
              char var;
              int value;} env_def;
}
```

Ex 2 – simple language

- Define a parser accepting

```
BEGIN MyProgram
  VARDEC   INT a,b,c
           FLOAT z,y;
  a = 4;
  b = z;
  y= c;
END MyProgram
```

- and providing functionality to build the parsing tree

Ex 2 – simple language

- The goal is to produce the parsing tree and not performing operations like +,*, …
  - it is the **translation** of the input code

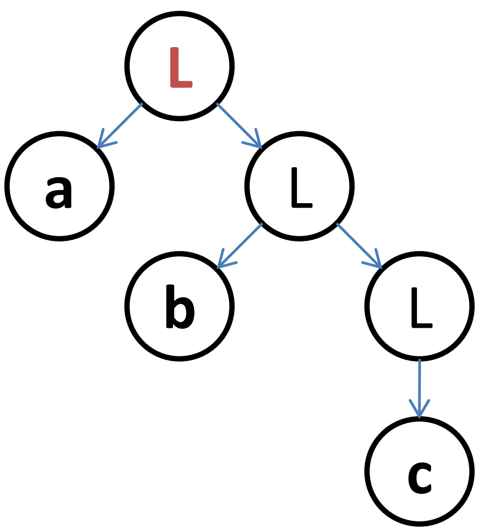
```
BEGIN MyProgram
  VARDEC   INT a,b,c


    a = 4;
END MyProgram
```

```
16. Program
  15. Body
    6. VarDec
      5. VarList
        4. NameList <- Name
          1. Name <- ID a
          2. Name <- ID b
          3. Name <- ID c
    14. StmtList <- Stmt
      13. …
```

Ex 2 – simple language

- Recursive rule (e.g., list) are represented only by the parent

VARDEC INT a,b,c



16. Program
  15. Body
    6. VarDec
      5. VarList
        4. **NameList**
          1. Name <- ID a
          2. Name <- ID b
          3. Name <- ID c
    14. StmtList <- Stmt
      13. …

Ex 2 – simple language

Program $\rightarrow$ BEGIN ProgramName Body END ProgramName

Body $\rightarrow$ VarDec ';' StatementList

VarDec $\rightarrow$ VARDEC VarList

VarList $\rightarrow$ Type NameList
VarList $\rightarrow$ Type NameList VarList

NameList $\rightarrow$ Name
NameList $\rightarrow$ Name ',' NameList

StatementList $\rightarrow$ Statement ';'
StatementList $\rightarrow$ Statement ';' StatementList

Ex 2 – simple language

Statement $\rightarrow$ Assignment

Assignment $\rightarrow$ Name '=' Expr

Expr $\rightarrow$ Term '+' Expr
Expr $\rightarrow$ Term
Term $\rightarrow$ Term '*' Factor
Term $\rightarrow$ Factor
Factor $\rightarrow$ '(' Expr ')' | NUMBER |Name

ProgramName $\rightarrow$ IDENTIFIER
Name $\rightarrow$ IDENTIFIER
Type $\rightarrow$ INT | FLOAT

Ex 2 - analysis

- The parsing tree (syntaxTree.h)

```
typedef struct _stNode{
  char *name;
  int nChildren;
  struct _child *children;
} stNode;

typedef struct _child{
  stNode *to;
  struct _child *next;
} child;
```

Ex 2 - analysis

- Functions

```
/* Creates a new stNode ad updates its field "name" */
stNode *createStNode(char *name);

/* Appends chld as child of parent. */
void appendChild(stNode *parent, stNode *chld);

// recursive Pretty Printing
void recursivePP(stNode *root, int ind);
```

Ex 2 - analysis

- Functions

```
/* Creates a new stNode ad updates its field "name" */
stNode *createStNode(char *name);

/* Appends chld as child of parent. */
void appendChild(stNode *parent, stNode *chld);

// recursive Pretty Printing
void recursivePP(stNode *root, int ind);
```

Ex 2 – types

- Types
  - IDENTIFIER: char*
  - NUMBER: int
  - Others: stNode*

```
%union {
  int intval;
  char *str;
  struct _stNode *nodep;
}
```

Ex 2 – types

%token <str> IDENTIFIER

%token <intval> NUMBER

%token INT

%token FLOAT

%token bEGIN

%token END

%token VARDEC

%type <nodep> Body

%type <nodep> StatementList

%type <nodep> VarDec

%type <nodep> VarList

%type <nodep> NameList

%type <nodep> Name

%type <nodep> Statement

%type <nodep> Assignment

%type <nodep> Expr

%type <nodep> Term

%type <nodep> Factor

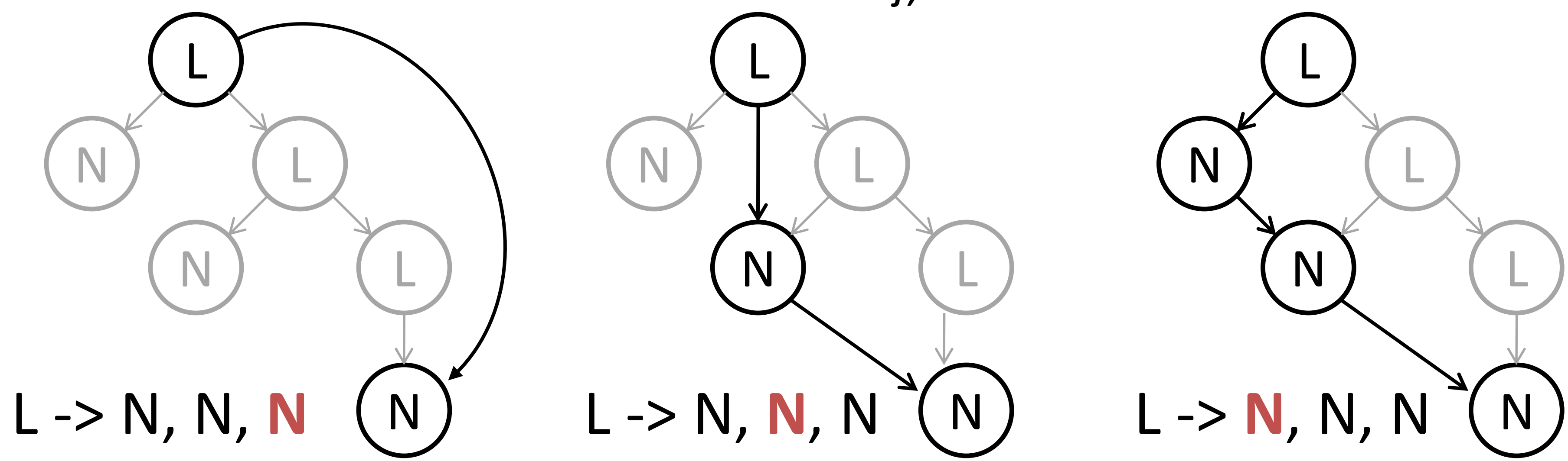
Ex 2 – build the syntax tree

```
Name
    : IDENTIFIER
    {
        sprintf(s, "%d. Name <- ID %s", i, $1);
        $$ = createStNode(strdup(s));
        printf("%d. Name <- ID %s \n", i, $1);
        i=i+1;
        free($1);
    }
    ;
```

```
[a-zA-Z]+ {
        yylval.str = strdup(yytext);
        return IDENTIFIER;
    }
```

Ex 2 – syntax tree of lists

```
NameList : Name                     {  $$ = createStNode(strdup(s));
                                       appendChild($$,$1);
                                    }
           | Name',' NameList          {  appendChild($3,$1);
                                          $$ = $3;
                                       };
```



Ex2 – parsing result

```
BEGIN MyProgram
 VARDEC
  INT a,b,c;

 a = c;

END MyProgram
```

1. Name <- ID a
2. Name <- ID b
3. Name <- ID c
4. NameList <- Name
5. NameList <- Name, NameList
6. NameList <- Name, NameList
7. VarList
8. VarDec
9. Name <- ID a
10. Name <- ID c
11. Factor <- Name
13. Term <- Factor
13. Expr <- Term
14. Assignment <- =
15. Stmt <- Assignment
16. StmtList <- Stmt
17. Body
18. Program

```
18. Program
 17. Body
  8. VarDec
   7. VarList
    4. NameList <- Name
     1. Name <- ID a
     2. Name <- ID b
     3. Name <- ID c
  16. StmtList <- Stmt
   15. Stmt <- Assignment
    14. Assignment <- =
     13. Expr <- Term
      12. Term <- Factor
       11. Factor <- Name
        10. Name <- ID c
     9. Name <- ID a
```