Lecture 6

Recursion

Two Types

Left recursion

Right recursion

Left Recursion

$A \rightarrow A\ \alpha \mid \beta$

Leads to **<u>Infinite loop</u>**

Generate the strings

$\beta\alpha$

$\beta\alpha\alpha\alpha\alpha$

Eliminate Left recursion

Make it right recursive

$A \rightarrow \beta\alpha^*$

Eliminating left recursion

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

Examples

1. E → E + T | T

2. S → SA | B

3. S → (L) | x

   L → L, S | S

Deterministic and Non-deterministic grammar

$A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid \alpha\ \beta_3$

Suppose, derive the string $\alpha\ \beta_3$

Backtracking

Non-deterministic grammar into deterministic grammar

$A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid \alpha\ \beta_3$

Use Left Factoring

$A \rightarrow \alpha\ A'$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$

Examples

1. $S \rightarrow AB \mid AC$
   $B \rightarrow b$
   $C \rightarrow c$

2. $S \rightarrow iEtS \mid iEtSeS \mid a$

3. $S \rightarrow aSSbS \mid aSaSb \mid abb \mid b$

Kindly go through the next slides..

Binding – 1/3

Each **program entity pe_i** in **program P** has a set of **attributes $A_i \equiv \{a_j\}$** associated with it

If pe_i is identifier, it has an attribute *kind* whose value indicates whether it is a variable, procedure, or reserved identifier (keyword)

A variable has attributes like type, dimensionality, scope, memory address, etc

Eg. Type is an attribute of a variable. It is also a program entity with its own attributes (i.e. size)

Binding – 2/3

Values of attributes of a type should be determined some time before a language processor processes a declaration statement using that type

var : type;

Binding – 3/3

Binding is the association of an attribute of a program entity with a value

Binding time is the time at which a binding is performed

The type attribute of variable *var* is bound to *typ* when its declaration is processed

The size attribute of *typ* is bound to a value sometime prior to this binding

Binding Times – 1/5

We are interested in following binding times

- Language definition time of L
- Language implementation time of L
- Compilation time of P
- Execution init time of *proc*
- Execution time of *proc*

Where L is prog. Lang, P is program written in L and *proc* is procedure in P

Binding Times – 2/5

Language definition time of L specifies binding times for attributes of various entities of a program written in L

Language implementation time is time when a language translator is designed

Binding Times – 3/5

```
program bindings (input, output);
        var
                i : integer;
                a, b : real;
        procedure proc (x : real; j : integer);
                var
                        info : array[1..10, 1..5] of integer
                        p : ↑integer;
                begin
                        new (p);
                end;
                begin
                        proc (a,i);
                end
```

Binding Times – 4/5

Binding of the **keywords** of Pascal to their meanings is performed at **language definition time**

At **language implementation time**, compiler designer performs certain bindings

Eg. **Size of type integer is bound to n bytes** where n is no. determined by architecture of target machine

Binding Times – 5/5

Binding of **type** attributes of **variables** is performed at **compilation time** of program *bindings*

Memory address of local variables *info* and *p* of procedure *proc* are bound at every **execution init time** of procedure proc

Importance of binding times – 1/4

Binding time of attribute of a program entity determines the manner in which a language processor can handle the use of entity

Compiler cannot generate such code for bindings performed later than compilation time

Importance of binding times – 2/4

Consider PL/1 program

```
procedure pl1_proc (x , j, info_size,        columns);
        declare x float;
        declare (j, info_size, columns) fixed;
        declare pl1_info (1: info_size, 1: columns) fixed;
        ….
end pl1_proc;
```

Importance of binding times – 3/4

The size of array *pl1_info* is determined by values of parameters *info_size* and *columns* in a specific call of *pl1_proc*

This is an instance of **execution time binding**

Compiler does not know the size of array *pl1_info*

Hence it may not be able to generate code for accessing its elements

Importance of binding times – 4/4

Dimension bounds of array *info* in program *bindings* are constants

Binding of dimension bound attributes can be performed at **compilation time**

This enables Pascal compiler to generate efficient code to access elements of info

Static and dynamic bindings

Static binding is a binding performed before execution of a program begins

Dynamic binding is a binding performed after execution of a program has begun

Static bindings lead to more efficient execution of a program than dynamic bindings

Language processor development tools - 1/8

Analysis phase of a language processor has a standard form irrespective of its purpose, source text is subjected to lexical, syntax and semantic analysis and results of analysis are represented in IR

This has led to development of a set of **language processor development tools (LPDTs)** focusing on **generation of analysis phase** of language processors

Language processor development tools - 2/8

Fig next shows a schematic of LPDT which generates the analysis of a language processor whose source language is L

The LPDT requires following two inputs

Specification of a grammar of language L

Specification of semantic actions to be performed in analysis phase

Language processor development tools - 3/8

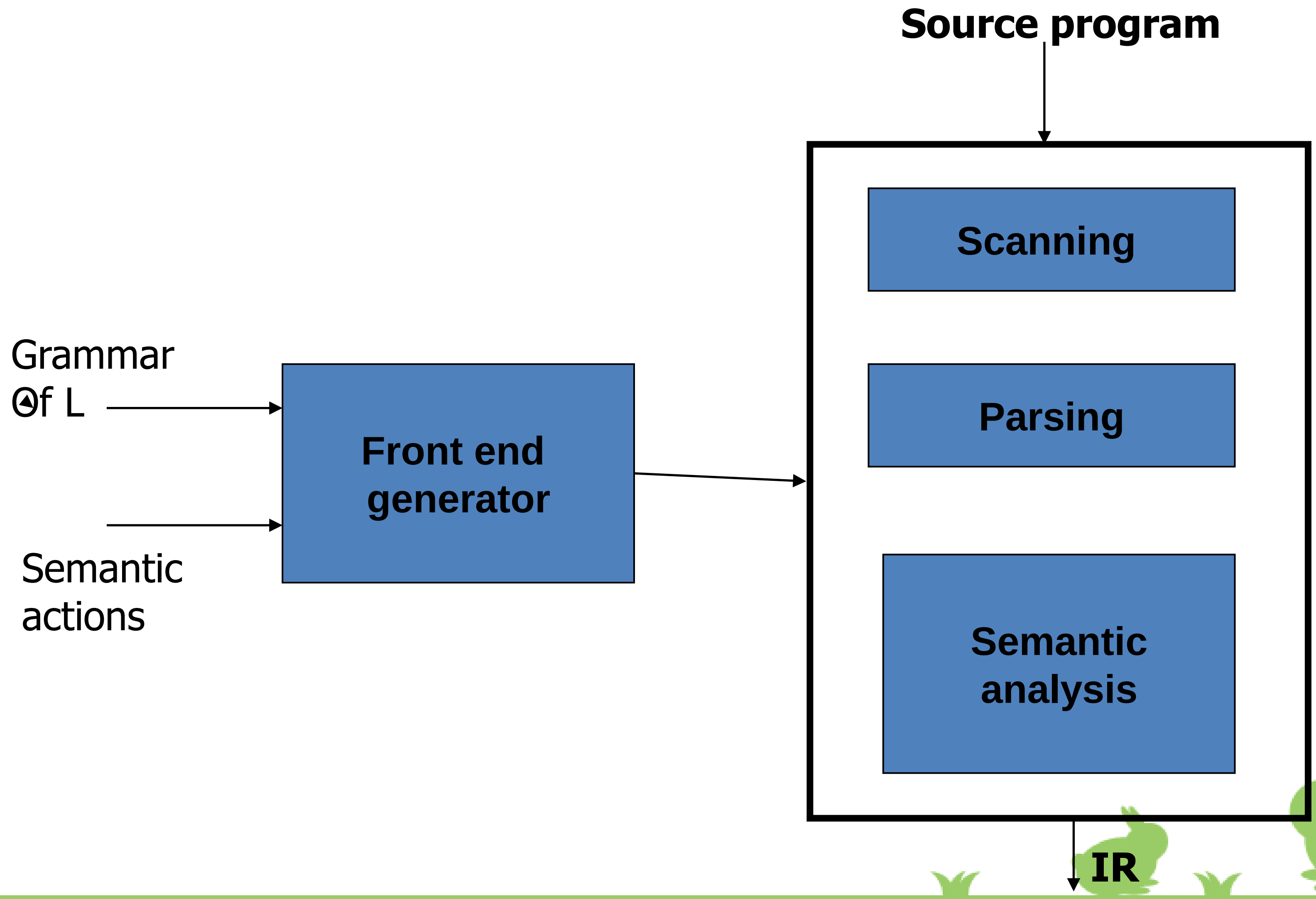

Fig: A language processor development tool (LPDT)

Language processor development tools - 4/8

It generates programs that perform lexical, syntax and semantic analysis of source program and construct IR

These programs collectively form the analysis phase of language processor

Two LPDTs used widely in practice

Lexical analyser generator LEX

Parser generator YACC

Language processor development tools -5/8

Input to these tools is a specification of lexical and syntactic constructs of L, and semantic actions to be performed on recognizing the constructs

Specification consists of a set of translation rules of the form

<string specification> {<semantic action>}

Where <semantic action> consists of C code

Language processor development tools - 6/8

This code is executed when a string matching **<string specification>** is encountered in i/p

LEX and YACC generate C programs which contain the code for scanning and parsing and semantic actions contained in specification

A YACC generated parser can use a LEX generated scanner as a routine if scanner and parser use **same conventions** concerning representation of tokens

Language processor development tools - 7/8

Fig next shows a schematic for developing analysis phase of compiler for language L using LEX and YACC

Analysis phase processes the source program to build an IR

A single pass compiler can be built using LEX and YACC is semantic actions are aimed at generating target code instead of IR

The scanner also generates IR of a SP for use by parser

Call it IR_l to differentiate it from IR of analysis phase

Language processor development tools - 8/8

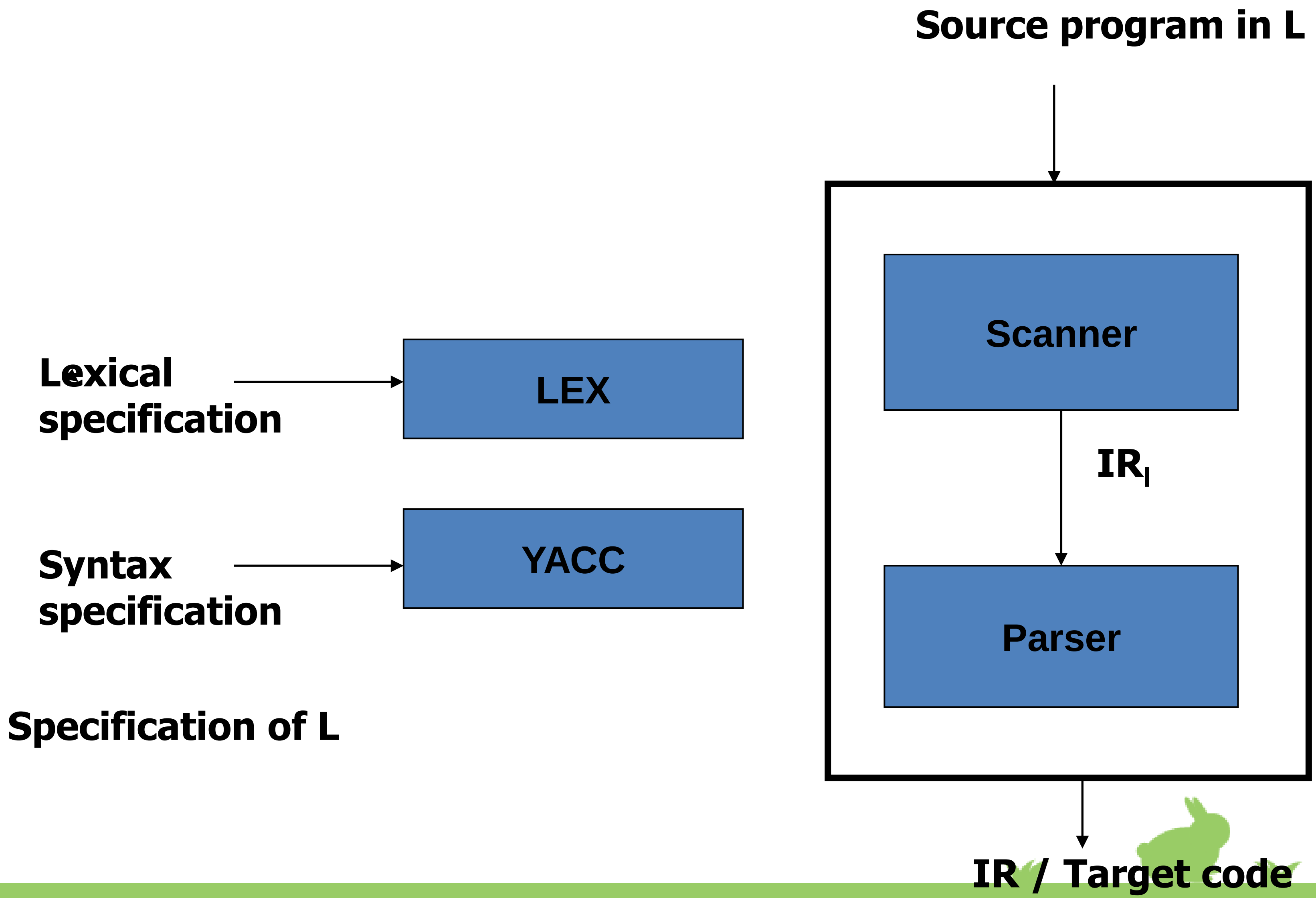

Fig: Using LEX and YACC

LEX – 1/5

LEX accepts i/p specification which consists of **two components**

Specification of strings representing lexical units in L. eg. Id's and constants.

Specification of semantic actions aimed at building IR. This consists of table of a set of tables of lexical units and a sequence of tokens for lexical units occurring in a source stmt. Semantic actions make new entries in tables and build tokens for lexical units

LEX – 2/5

```
%{
letter                                  [A-Za-z]
digit                                   [0-9]
}%

%%
begin                                   {return(BEGIN);}
end                                              {return(END);}
“:=”                                             {return(ASGOP);}
{letter} ({letter}|{digit})*            {yylval=enter_id();
                                                 return (ID);}
{digit}+                                {yylval=enter_num();
                                                 return (NUM);}
%%
enter_id()
{ /* enters the id in the symbol table and returns entry number */}
enter_num()
{ /* enters the number in the constants table and returns entry number
*/}
```

LEX – 3/5

The i/p consists of **four components**, three of which are shown

First component (enclosed by *%{ and %}*)
Defines the **symbols used** in specifying the strings of L

Second component (enclosed by *%% and %%)*
Contains **translation rules**

Third component
Contains **auxiliary routines** which can be used in semantic action

LEX – 4/5

Sample i/p defines the strings **begin**, **end**, **:=** (assignment operator), and **identifier** and **constant** strings of L

When identifier is found, it is entered in symbol table using routine *enter_id*

The pair ***(ID, entry#)*** **forms the token** for identifier string

LEX – 5/5

The *entry#* is put in global variable *yylval*, and class **code ID is returned** as value of the call on scanner

Similar actions are taken on finding a constant, keywords *begin* and *end* and assignment operator

YACC – 1/5

Each string specification in i/p to YACC resembles grammar production

The parser generated by YACC performs reductions according to this grammar

An **attribute** is associated with **every nonterminal symbol**

Value of this attribute can be manipulated during parsing

YACC – 2/5

Attribute can be given any **user-designed structure**

A symbol ‘**$n**’ in action part of a translation rule **refers to attribute of nth symbol in RHS** of the string specification

‘$$’ represents the **attribute of LHS symbol of string specification**

YACC – 3/5

```
%%
E : E+T         {$$ = gencode('+', $1, $3);}
   | T  {$$ = $1;}
  ;
T : T+V         {$$ = gencode('*', $1, $3);}
   | V  {$$ = $1;}
   ;
V : id          {$$ = gencode($1);}
   ;
%%
```

gencode (operator, operand_1, operand_2)
(/* Generates code using operand descriptors. Returns descriptor for result */}
gendesc (symbol)
(/* Refer to symbol/ constant table entry. Build and return descriptor for the symbol*/}

YACC – 4/5

i/p consists of **four components**, two of which are shown

The routine **gendesc** builds a descriptor containing name and type of id or constant

Routine **gencode** takes an operator and attributes of two operands, generates code and returns with attribute for result of operation

YACC – 5/5

Parsing of **string b+c*d** where **b, c and d are of type real**, using the parser generator by YACC from i/p before leads to following calls on C routines

Gendesc (Id #1);
Gendesc (Id #2);
Gendesc (Id #3);
Gencode (*, *c,real*, *d,real*);
Gencode (+, *b,real*, *t,real*);

where attribute has the form <*name*>,<*type*> and *t* is name of location (a register or memory word) used to store result of c*d in code generated by first call on *gencode*