Yacc

Lexical vs. Syntactic Analysis

Phase	**Input**	**Output**
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	Parse tree

- **`Lex`** is a tool for writing lexical analyzers.
- **`Yacc`** is a tool for constructing parsers.

Slide credit: Wes Weimer

The Functionality of the Parser

- Input: sequence of tokens from lexer

- Output: parse tree of the program
  - Also called an abstract syntax tree
- Output: error if the input is not valid
  – e.g., “parse error on line 3"

Example

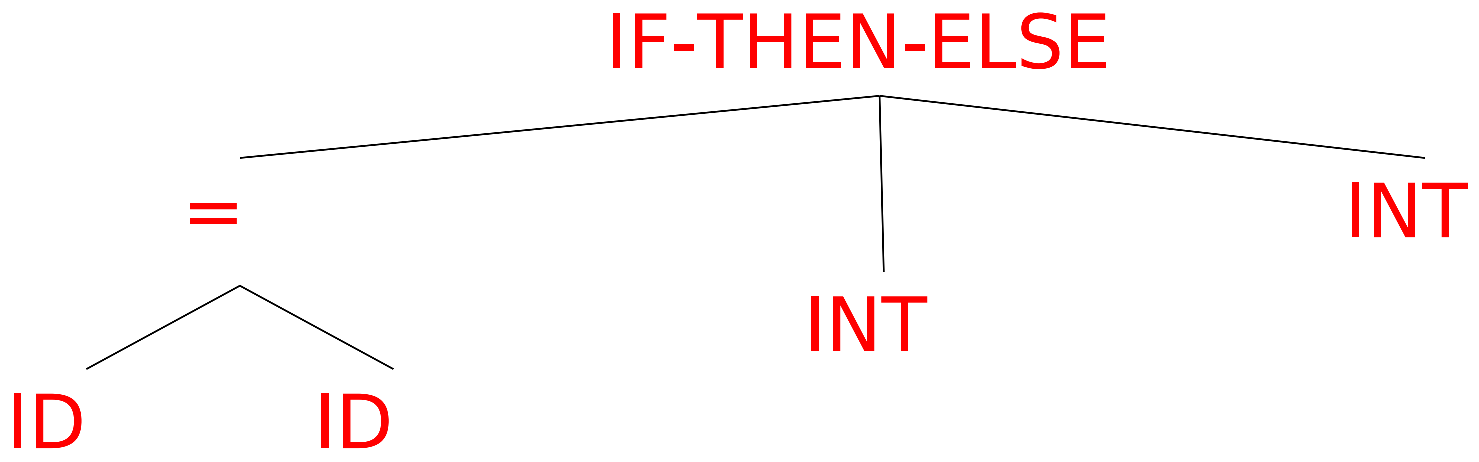
- Cool program text

```
if x = y then 1 else 2 fi
```

- Parser input (tokens)

  IF ID = ID THEN INT ELSE INT FI

- Parser output (tree)



The Role of the Parser

- Not all sequences of tokens are programs
  - `then x * / + 3 while x ; y z then`
- The parser must distinguish between valid and invalid sequences of tokens
  - We need context free grammars.

- **`Yacc`** stands for <u>**y**</u>et <u>**a**</u>nother <u>**c**</u>ompiler to <u>**c**</u>ompiler.
  - Reads a specification file that codifies the grammar of a language and generates a parsing routine

YACC – Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories,
Murray Hill, New Jersey 07974

ABSTRACT

Computer program input generally has some structure; in fact, every computer program which does input can be thought of as defining an ‘‘input language’’ which it accepts. The input languages may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, standard input facilities are restricted, difficult to use and change, and do not completely check their inputs for validity.

Yacc provides a general tool for controlling the input to a computer program. The Yacc user describes the structures of his input, together with code which is to be invoked when each such structure is recognized. Yacc turns such a specification into a subroutine which may be invoked to handle the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user´s application handled by this subroutine.

The input subroutine produced by Yacc calls a user supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or, if he wishes, in terms of higher level constructs such as names and numbers. The user supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy specification.

Yacc is written in C[7], and runs under UNIX. The subroutine which is output may be in C or in Ratfor[4], at the user´s choice; Ratfor permits translation of the output subroutine into portable Fortran[5]. The class of specifications accepted is a very general one, called LALR(1) grammars with disambiguating rules. The theory behind Yacc has been described elsewhere[1,2,3].

Yacc was originally designed to help produce the ‘‘front end’’ of compilers; in addition to this use, it has been successfully used in many application programs, including a phototypesetter language, a document retrieval system, a Fortran debugging system, and the Ratfor compiler.

1970

`Yacc`

`Yacc` specification describes a Context Free Grammar (CFG), that can be used to generate a parser.

Elements of a CFG:

1. Terminals: tokens and literal characters,
2. Variables (nonterminals): syntactical elements,
3. Production rules, and
4. Start rule.

Yacc

- Format of a production rule:

```
symbol: definition
{action}
;
```

Example:

A $\rightarrow$ **Bc** is written in **`yacc`** as **`a: b 'c';`**

Yacc Format

- Format of a **`yacc`** specification file:

```
declarations
%%
grammar rules and associated actions
%%
C programs
```

Running **yacc** on Linux

In Linux there is no **liby.a** library for **yacc** functions

You have to add the following lines to end of your **yacc** specification file

```
int yyerror(char *s)
{
 printf(“%s\n”, s);
}
int main(void)
{
 yyparse();
}
```

Then type

```
gcc -o exe_file y.tab.c -lfl
```

```
/*lex */
%%
.|\n { return yytext[0];}
```

```
/*yacc */
%%
start:rule '\n' {printf("I RECOGNIZED ANBN"); return 0;}
rule: 'a' 'b';
%%
#include "lex.yy.c"
int yyerror(char *s)
{  printf("%s, DOES NOT MATCH TO ANBN\n", s); }
int main(void)
{ yyparse(); }
```

```
/*lex */
%%
.|\n { return yytext[0];}
```

```
/*yacc */
%%
start:rule '\n' {printf("I RECOGNIZED ANBN"); return 0;}
rule:'a''b'|'a'rule'b' ;
%%
#include "lex.yy.c"
int yyerror(char *s)
{   printf("%s, DOES NOT MATCH TO ANBN\n", s)}
int main(void)
{  yyparse();}
```

Declarations

To define tokens and their characteristics

`%token:` declare names of tokens

A simple yacc specification to accept $L=\{ a^n b^n \mid n>=1\}$.

```
/*anbn0.y */
%token A B
%%
start: anbn '\n' {return 0;}
anbn: A B
| A anbn B
;
%%
#include "lex.yy.c"
```

lex – yacc pair

```
/* anbn0.l */
%%
a return (A);
b return (B);
. return (yytext[0]);
\n return ('\n');
```

```
/*anbn0.y */
%token A B
%%
start: anbn '\n' {return 0;}
anbn: A B
| A anbn B
;
%%
#include "lex.yy.c"
```

Printing messages

If the input stream does not match **`start`**, the default message of **`"syntax error"`** is printed and program terminates.

However, customized error messages can be generated.

```
/*anbn1.y */
%token A B
%%
start: anbn '\n' {printf(" is in anbn\n");
                   return 0;}
anbn: A B
| A anbn B
;
%%
#include "lex.yy.c"
int yyerror(char *s)
{printf("%s, it is not in anbn\n", s);}
int main(void)
{  yyparse();}
```

Example Output

```
$anbn
aabb
is in anbn
$anbn
acadbefbg
Syntax error, it is not in anbn
```

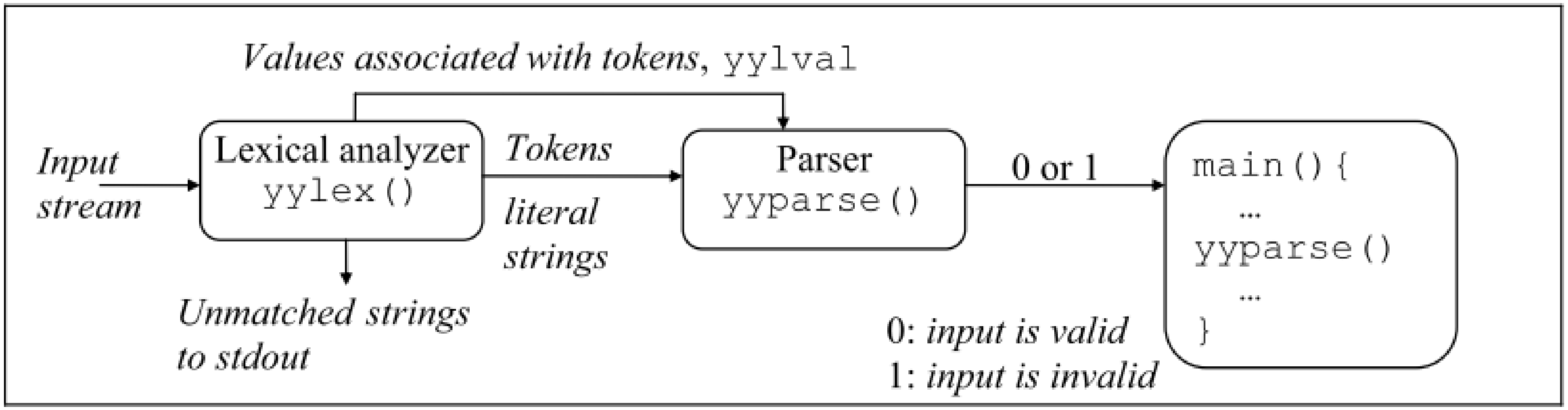
A grammar to accept L = {$a^n b^n$ | n >= 0}.

```
/*anbn_0.y */
%token A B
%%
start: anbn '\n' {printf(" is in anbn_0\n");
                  return 0;}
anbn: empty
| A anbn B
;
empty: ;
%%
#include "lex.yy.c"
int yyerror(char *s)
{printf("%s, it is not in anbn\n", s);}
int main(void)
{  yyparse();}
```

Recursive Rules

Although right-recursive rules can be used in **`yacc`**, left-recursive rules are preferred, and, in general, generate more efficient parsers.

yylval



`yylex()` function returns an integer, the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable **`yylval`**.

Yacc Format

- Format of a **`yacc`** specification file:

```
declarations
%%
grammar rules and associated actions
%%
C programs
```

Declarations

To define tokens and their characteristics

`%token:` declare names of tokens

`%left:` define left-associative operators

`%right:` define right-associative operators

`%nonassoc:` define operators that may not associate with themselves

`%type:` declare the type of variables

Declarations

`%union:` declare multiple data types for semantic values

`%start:` declare the start symbol (default is the first variable in rules)

`%prec:` assign precedence to a rule

`%{`

`C declarations` directly copied to the resulting C program

`%}` (e.g., variables, types, macros…)

yylval

The type of **`yylval`** is **`int`** by default. To change the type of **`yylval`** use macro **`YYSTYPE`** in the declarations section of a **`yacc`** specifications file.

```
%{
#define YYSTYPE double
%}
```

If there are more than one data types for token values, **`yylval`** is declared as a **`union`**.

yylval

Example with three possible types for **`yylval`**:

```
%union{
double real; /* real value */
int integer; /* integer value */
char str[30]; /* string value */
}
```

Example:

`yytext=“0012”`, type of **`yylval:int`**, value of **`yylval: 12`**

`yytext=“+1.70”`, type of **`yylval:float`**, value of **`yylval:1.7`**

Token types

- The type of associated values of tokens can be specified by **`%token`** as

```
%token <real> REAL
%token <integer> INTEGER
%token <str> IDENTIFIER STRING
```

- Type of variables can be defined by **`%type`** as

```
%type <real> real-expr
%type <integer> integer-expr
```

To return values for tokens from a lexical analyzer:

```
/* lexical-analyzer.l */
alphabetic [A-Za-z]
digit [0-9]
alphanumeric ({alphabetic}|{digit})
%%
[+-]?{digit}*(\.)?{digit}+ {sscanf(yytext, "%lf",
                             &yylval.real);
                             return REAL;
                            }
{alphabetic}{alphanumeric}* {strcpy(yylval.str,yytext);
                             return IDENTIFIER;
                             }
```

Positional assignment of values for items

$$: left-hand side

$1: first item in the right-hand side

$n: nth item in the right-hand side

Example: Printing integers

```
/*print-int.l*/
%%
[0-9]+ {sscanf(yytext, "%d", &yylval);return(INTEGER);}
\n return(NEWLINE);
. return(yytext[0]);
```

```
/* print-int.y */
%token INTEGER NEWLINE
%%
lines: /* empty */
| lines NEWLINE
| lines line NEWLINE {printf("=%d\n", $2);}
| error NEWLINE {yyerror("Reenter:"); yyerrok;}
;
line: INTEGER {$$ = $1;}
;
%%
#include "lex.yy.c"
```

Yacc provides a special symbol for handling errors. The symbol is called `error` and it should appear within a grammar-rule.

Example continued

```
$print-int
7
=7
007
=7
zippy
syntax error
Reenter:

_
```

Operator Precedence

- All of the tokens on the same line are assumed to have the same precedence level and associativity;
- The lines are listed in order of increasing precedence or binding strength.

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

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative.

Example: simple calculator - `lex`

```
/* calculator.l */
integer         [0-9]+
dreal           ([0-9]*\.[0-9]+)
ereal           ([0-9]*\.[0-9]+[Ee][+-]?[0-9]+)
real            {dreal}|{ereal}
nl              \n
%%
[ \t]           ;
{integer}       { sscanf(yytext, "%d", &yylval.integer);
                  return INTEGER;
                }
{real}          { sscanf(yytext, "%lf", &yylval.real);
                  return REAL;
                }
\+              { return PLUS;}
\-              { return MINUS;}
\*              { return TIMES;}
\/              { return DIVIDE;}
\(              { return LP;}
\)              { return RP;}
{nl}            { extern int lineno; lineno++;
                  return NL;
                }
.               { return yytext[0]; }
%%
int yywrap() { return 1; }
```

Example: simple calculator - `yacc`

```
/* calculator.y */
%{
#include <stdio.h>
%}
%union{ double   real; /* real value */
        int    integer; /* integer value */
      }
%token <real> REAL
%token <integer> INTEGER
%token PLUS MINUS TIMES DIVIDE LP RP NL
%type <real> rexpr
%type <integer> iexpr
%left PLUS MINUS
%left TIMES DIVIDE
%left UMINUS
```

Example: simple calculator - yacc

```
%%
lines: /* nothing */
     | lines line
     ;
line:  NL
     | iexpr NL
       { printf("%d) %d\n", lineno, $1);}
     | rexpr NL
       { printf("%d) %15.8lf\n", lineno, $1);}
     ;
iexpr: INTEGER
     | iexpr PLUS iexpr
       { $$ = $1 + $3;}
     | iexpr MINUS iexpr
       { $$ = $1 - $3;}
     | iexpr TIMES iexpr
       { $$ = $1 * $3;}
     | iexpr DIVIDE iexpr
       { if($3) $$ = $1 / $3;
         else { yyerror("divide by zero");      }
       }
     | MINUS iexpr %prec UMINUS
       { $$ = - $2;}
     | LP iexpr RP
       { $$ = $2;}
     ;
```

Example: simple calculator - yacc

```
rexpr: REAL
     | rexpr PLUS rexpr
       { $$ = $1 + $3;}
     | rexpr MINUS rexpr
       { $$ = $1 - $3;}
     | rexpr TIMES rexpr
       { $$ = $1 * $3;}
     | rexpr DIVIDE rexpr
       { if($3) $$ = $1 / $3;     else { yyerror( "divide by zero" );    }    }
     | MINUS rexpr %prec UMINUS
       { $$ = - $2;}
     | LP rexpr RP
       { $$ = $2;}
     | iexpr PLUS rexpr
       { $$ = (double)$1 + $3;}
     | iexpr MINUS rexpr
       { $$ = (double)$1 - $3;}
     | iexpr TIMES rexpr
       { $$ = (double)$1 * $3;}
     | iexpr DIVIDE rexpr
       { if($3) $$ = (double)$1 / $3;  else { yyerror( "divide by zero" );   }
   }
     | rexpr PLUS iexpr
       { $$ = $1 + (double)$3;}
     | rexpr MINUS iexpr
       { $$ = $1 - (double)$3;}
     | rexpr TIMES iexpr
       { $$ = $1 * (double)$3;}
     | rexpr DIVIDE iexpr
       { if($3) $$ = $1 / (double)$3;  else { yyerror( "divide by zero" );   }
   ;
```

Actions between Rule Elements

```
/* lex specification */
%%
a return A;
b return B;
\n return NL;
. ;
%%
int yywrap() { return 1; }
```

```
/* yacc specification */
%{
#include <stdio.h>
%}
%token A B NL
%%
s: {printf("1");}
   a
   {printf("2");}
   b
   {printf("3");}
   NL
   {return 0;}
   ;
a: {printf("4");}
   A
   {printf("5");}
   ;
b: {printf("6");}
   B
   {printf("7");}
   ;
%%
#include "lex.yy.c"
int yyerror(char *s) {
  printf ("%s\n", s);
}

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

input: `ab`
output: `1452673`

input: `aa`
output: `14526 syntax error`

input: `ba`
output: `14 syntax error`

References

- http://memphis.compilertools.net/interpreter.html
- http://www.opengroup.org/onlinepubs/007908799/xcu/yacc.html
- http://dinosaur.compilertools.net/yacc/index.html