About Yacc

Alberto Ercolani

University of Trento

alberto.ercolani@unitn.it

November 21, 2017

What is a parser?

- A parser is a PDA: LALR(1) automaton of a grammar $\mathcal{G}$.
- It runs the Shift Reduce algorithm to decide whether or not a string of symbols belongs to $\mathcal{L}(\mathcal{G})$.

What is a parser?

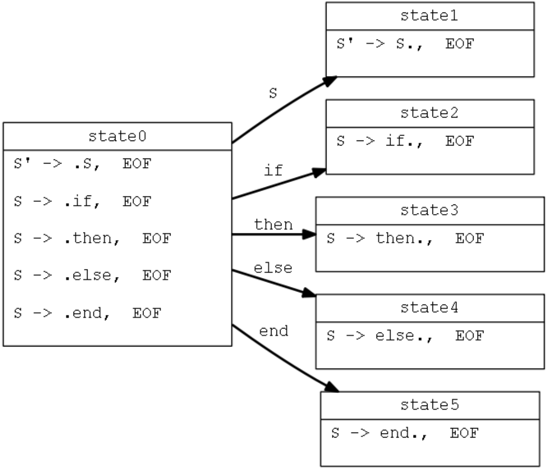


Figure: A simple LALR(1) parser accepting if|then|else|end. EOF is $ symbol.

Lex & Yacc

- Lex & Yacc are designed to work together.
- Nothing forbids them to work alone:
  - Lex can read and manage context but it's not comfortable.
  - Yacc can parse with limited reading capabilities producing impractical languages.
- Yacc alone can not do much because the yylex() function can return only one value: "int yylex()".
- That's why we need an intercommunication channel between lexer and parser.

Lexer/Parser communication

- As you know Shift Reduce algorithm involves the lexer to tell which terminal symbol we are currently reading.
- As the algorithm executes the lexer makes a lot of readings on the file.
- Parser and lexer behave through master-slave approach.
  - int iToken = Lexer.GetNextToken();.

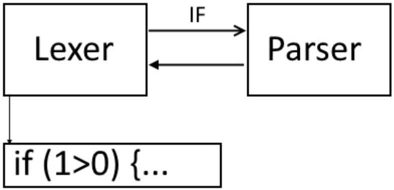


Figure: As shift reduce algorithm goes on the lexer tests different tokens.

Lexer/Parser communication

- As you know Shift Reduce algorithm involves the lexer to tell which terminal symbol we are currently reading.
- As the algorithm executes the lexer makes a lot of readings on the file.
- Parser and lexer behave through master-slave approach.
    - int iToken = Lexer.GetNextToken();.

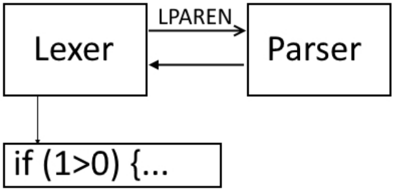


Figure: As shift reduce algorithm goes on the lexer tests different tokens.

Lexer/Parser communication

- As you know Shift Reduce algorithm involves the lexer to tell which terminal symbol we are currently reading.
- As the algorithm executes the lexer makes a lot of readings on the file.
- Parser and lexer behave through master-slave approach.
  - int iToken = Lexer.GetNextToken();.

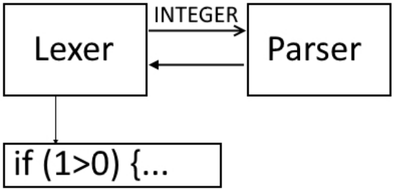


Figure: As shift reduce algorithm goes on the lexer tests different tokens.

Lexer/Parser communication

- As you know Shift Reduce algorithm involves the lexer to tell which terminal symbol we are currently reading.
- As the algorithm executes the lexer makes a lot of readings on the file.
- Parser and lexer behave through master-slave approach.
  - int iToken = Lexer.GetNextToken();.

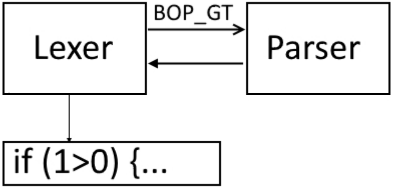


Figure: As shift reduce algorithm goes on the lexer tests different tokens.

Lexer/Parser communication

- As you know Shift Reduce algorithm involves the lexer to tell which terminal symbol we are currently reading.
- As the algorithm executes the lexer makes a lot of readings on the file.
- Parser and lexer behave through master-slave approach.
  - int iToken = Lexer.GetNextToken();.

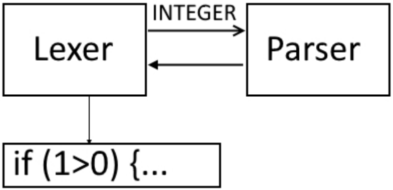


Figure: As shift reduce algorithm goes on the lexer tests different tokens.

What is Yacc?

- YACC is the acronym for Yet Another Compiler Compiler.
- Yacc is nowadays substituted by Bison.
- It generates parsers whose language is described through Context Free Grammars.
- Each production can be annotated by a list of valid C statements called "semantic action".
- In case the right hand side of a production matches a string of symbols in the language, a reduction happens and the corresponding semantic action is executed.
  - This makes the parser more powerful than a pushdown automaton!
  - E.G.: It can accept $\mathrm{L} = \{wcw | w \in (0|1)^+\}$.

What is Yacc?

- Yacc is a parser generator.
- Given the grammar of the *context free* language you want to match it generates C code recognizing its strings.
- You can build your own parser generator using the algorithms seen during the course!
  - SLR(1)/LALR(1)/LR(1) automaton construction.

What can Yacc do?

- Given a grammar $\mathcal{G}$, tells you whether or not it's LALR(1).
- Given some non-LALR(1) grammar gives you some tool to parse it.
- More (IELR(1), GLR), not part of this course.

Yacc's file structure

```
%{
/* This code is copied verbatim
into the parser's source code. */
%}
/* Yacc directives here.*/
%%
/* Grammar rules here.*/
%%
/* This section is copied verbatim
into the parser's source. */
```

- Rings a bell?

CFG grammars

$$
\begin{array}{lcll}
\mathtt{S} & \rightarrow & \mathtt{E} & \{\ldots\} \\
\mathtt{E} & \rightarrow & \mathtt{E + T} & \{\ldots\} \\
 & | & \mathtt{T} & \{\ldots\} \\
\mathtt{T} & \rightarrow & \mathtt{T * F} & \{\ldots\} \\
 & | & \mathtt{F} & \{\ldots\} \\
\mathtt{F} & \rightarrow & \mathtt{(E)} & \{\ldots\} \\
 & | & \mathtt{ID} & \{\ldots\}
\end{array}
$$

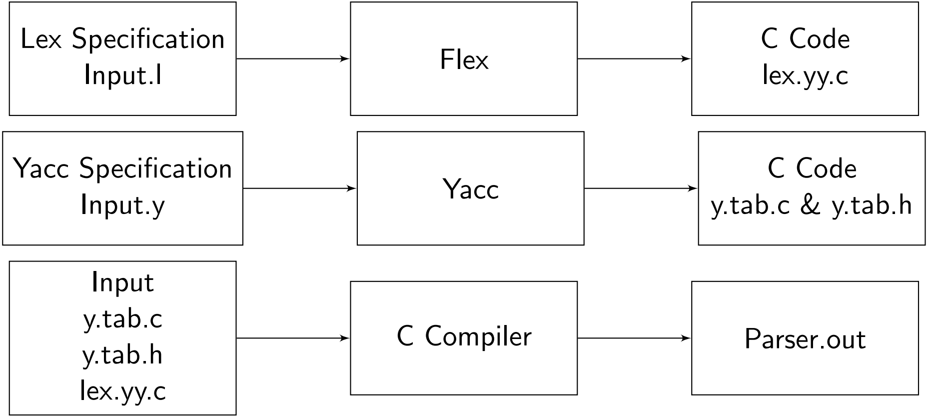
- This is a rather famous grammar.

Yacc's grammar rules

```
S:  E                                   {...};
E:  E '+' T                             {...}
  | T                                   {...};
T:  T '*' F                             {...}
  | F                                   {...};
F:  '(' E ')'                           {...}
  | ID                                  {...};
```

- As you can notice with minimum adjustments it can be made compatible with Yacc.

Generating/Compiling/Running a Parser



- flex Input.l
- bison -d input.y -o y.tab.c
- CC lex.yy.c y.tab.c -o Lexer.out -std=c99
- Parser.out < In.txt

Yacc directives customize your design

%start NonTerminal

- Sets S as start symbol, if missing the first symbol found is used.
- Example in folder “%start”.

- %token TerminalName_1 TerminalName_2 ... TerminalName_n
  - Declares terminal symbols: used in lexer semantic actions.
  - Example in folder “%token”.

- %union { (type Identifier)+ }
  - Declares a union of types and identifiers.
  - Example in folder “%union”.

- %type <Identifier> SymbolName_1 SymbolName_2 ... SymbolName_n
  - Attributes a type to nodes in semantic actions.
  - Example in folder “%type”.

Yacc directive: %start

- %start NonTerminal
  - When Yacc encounters this directive sets “NonTerminal” as starting symbol.

```
%start S

S: E                                {...};
E: E '+' T                          {...}
 | T                                {...};
T: T '*' F                          {...}
 | F                                {...};
F: '(' E ')'                        {...}
 | ID                               {...};
```

- Is there any way to spot the start symbol without assuming it is the first or pointing it out?

Yacc directives customize your design

- %start NonTerminal
  - Sets S as start symbol, if missing the first symbol found is used.
  - Example in folder "%start".

%token TerminalName$_1$ TerminalName$_2$ ... TerminalName$_n$

- Declares terminal symbols: used in lexer semantic actions.
- Example in folder "%token".

- %union { (type Identifier)+ }
  - Declares a union of types and identifiers.
  - Example in folder "%union".

- %type <Identifier> SymbolName$_1$ SymbolName$_2$ ... SymbolName$_n$
  - Attributes a type to nodes in semantic actions.
  - Example in folder "%type".

Yacc directive: %token

- %token TN1 TN2 ... TNn
  - When Yacc encounters this directive defines in y.tab.h a set of integers, each one associated with a terminal.

```
[Inside Yacc Specification: grammar.y]
%token BOOL INT FLOAT

[Inside Lex Specification: lexer.l]
%{
#include "y.tab.h"
%}
%%
[0-9]*\.[0-9]+   {return FLOAT;}
[0-9]+           {return INT;}
"true"           {return BOOL;}
```

- Without %token directive terminals agreement should be done by hand.

Yacc directives customize your design

- %start NonTerminal
  - Sets S as start symbol, if missing the first symbol found is used.
  - Example in folder "%start".

- %token TerminalName_1 TerminalName_2 ... TerminalName_n
  - Declares terminal symbols: used in lexer semantic actions.
  - Example in folder "%token".

%union { (type Identifier)+ }

- Declares a union of types and identifiers.
- Example in folder "%union".

- %type <Identifier> SymbolName_1 SymbolName_2 ... SymbolName_n
  - Attributes a type to nodes in semantic actions.
  - Example in folder "%type".

What is a C/C++ union?

- union { /* Valid list of C identifiers. */ };
  - A special directive associating many identifiers to the same memory region.
  - The size of the union is the size of the largest item.
  - Any identifier can be referred and edited.
  - Correct result depend on bytes interpretation.

What is a C/C++ union?

```
struct Test {
        union {
                int iFirst;
                char cSecond;
                char cThird[2];
                void * pPointer;
        };
};
struct Test *a = ... /* Allocate empty Test.*/;
a->pPointer = 0x000000DD;          //1.
a->iFirst += 0xFF00CC00;           //2.
a->cSecond = 0xAA;                 //3.
a->cThird[1] = 0xBB;               //4.
```

- Assume compilation on a 32 bit machine, how big is Test structure?
- After operation 4. the value inside struct Test*a is equal to 0xAABBCCDD, do you agree?

Yacc directive: %union

- %union { /* Valid list of C identifiers. */ }
  - When Yacc encounters this directive it creates a union, used as bridge between lexer and parser.

```
[Inside Yacc Specification: grammar.y]
%token BOOL FLOAT INT
%union { int iValue, bool bValue, float fValue; }

[Inside Lex Specification: lexer.l]
%{#include "y.tab.h"
%}
%%
[0-9]*\.[0-9]+ {yyvalue.fValue = ... ;return FLOAT;}
[0-9]+  {yylval.iValue = ... ;return INT;}
"true"  {yylval.bValue = true;return BOOL;}
"false" {yylval.bValue = false;return BOOL;}
```

Yacc directives customize your design

- %start NonTerminal
  - Sets S as start symbol, if missing the first symbol found is used.
  - Example in folder "%start".

- %token TerminalName$_1$ TerminalName$_2$ ... TerminalName$_n$
  - Declares terminal symbols: used in lexer semantic actions.
  - Example in folder "%token".

- %union { (type Identifier)+ }
  - Declares a union of types and identifiers.

%type <Identifier> SymbolName$_1$ SymbolName$_2$ ... SymbolName$_n$

- Attributes a type to nodes in semantic actions.
- Example in folder "%type".

Yacc directive: %type

- %type <Identifier> TN1 TN2 ... TNn
  - This directive relates identifiers to terminals/non terminals.
  - Identifier is a valid identifier declared in %union directive.

Yacc directive: %type example

```
[Inside Yacc Specification: grammar.y]
%token INT ID
%union { int iValue, char *cString; }
%type <iValue> E T INT
%type <cString> ID

E: E+T  { $$ = $1 + $3;}
 | T    { $$ = $1;};
T: ID   { $$ = GetValue($1); }
 | INT  { $$ = $1;};

[Inside Lex Specification: lexer.l]
%{#include "y.tab.h"%}
%%
[0-9]+          {yylval.iValue = ... ;return INT;}
[a-zA-Z]        {yylval.cString = ...;return ID;}
```

Indexing tree's data structure

```
B:  B 'or' O           {$$ = $1 || $3;}
  | O                  {$$ = $1;}
O:  O 'and' A          {$$ = $1 && $3;}
  | A                  {$$ = $1;}
A:  'not' G            {$$ = !$1;}
  | G                  {$$ = $1;}
G:  '(' B ')'          {$$ = $2;}
  | 'true'             {$$ = true;}
  | 'false'            {$$ = false;}
```

- Access to data structures (children nodes of a Non Terminal symbol) is achieved through 1-based indexing.
- Index must be prefixed by $.

Indexing tree's data structure

```
B:  B 'or' O           {$$ = $1 || $3;}
  | O                  {$$ = $1;}
O:  O 'and' A          {$$ = $1 && $3;}
  | A                  {$$ = $1;}
A:  'not' G            {$$ = !$1;}
  | G                  {$$ = $1;}
G:  '(' B ')'          {$$ = $2;}
  | 'true'             {$$ = true;}
  | 'false'            {$$ = false;}
```

- As you are certainly imagining the $$ value is the parent of the nodes playing the role of symbols in the right hand side of productions.
- Which is the abstract syntax tree of "false implies true"?

Yacc directives customize your design

- %start NonTerminal
- %token $\mathsf{TerminalName}_1$ $\mathsf{TerminalName}_2$ ... $\mathsf{TerminalName}_n$
- %union { (type Identifier)+ }
- %type <Identifier> $\mathsf{SymbolName}_1$ $\mathsf{SymbolName}_2$ ... $\mathsf{SymbolName}_n$

Hands on!

- Design an unambiguous grammar for arithmetic expressions whose operations are: difference, sum, product, division, *exponentiation*.
- Remember to encode precedence and associativity:
  - Difference and sum are left associative.
  - Product, division and exponentiation are right associative.

Bibliography

D. Brown, J. Levine and T. Mason
Lex & Yacc, 2nd Edition
O'Reilly Media

J. Levine
Flex & Bison
O'Reilly Media