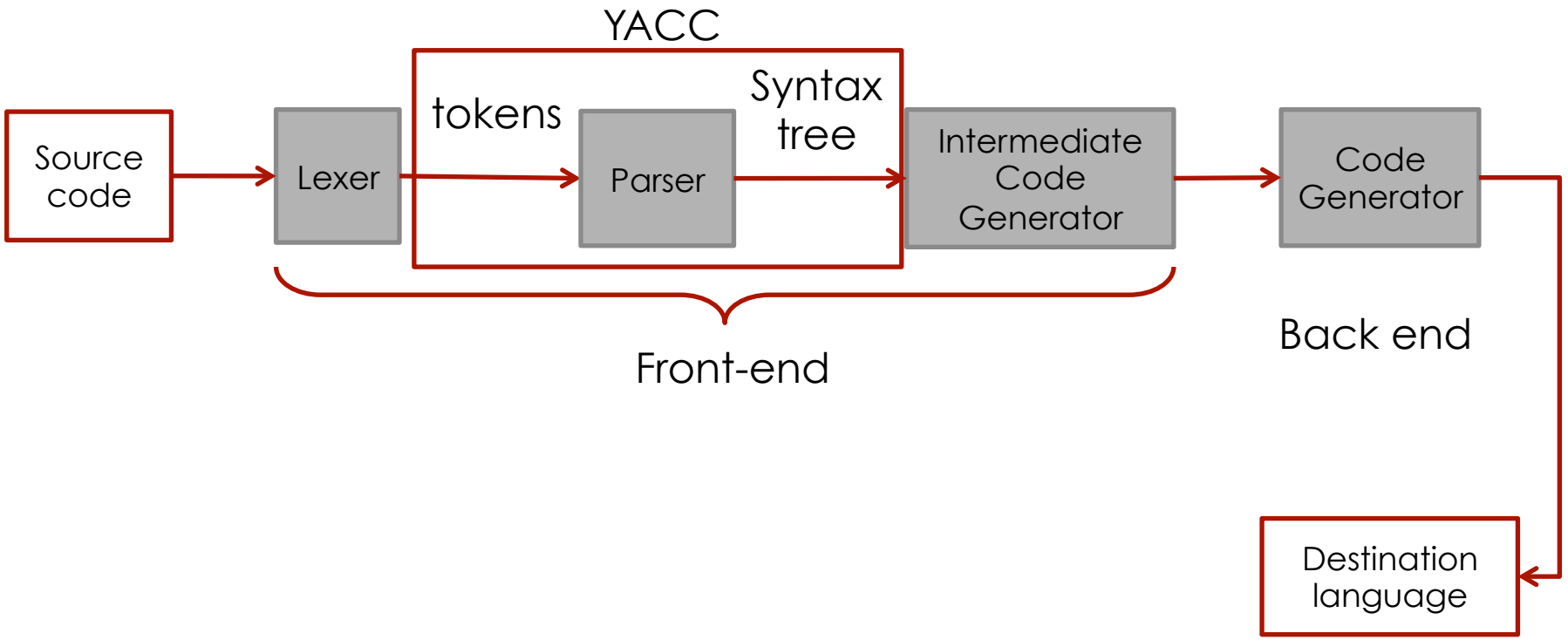
Formal Languages and Compilers

YACC

What is yacc?

- Yacc = Yet Another Compiler Compiler
- Describing the input of the source file→ define a structure of a program
- Describe what to do when such input is recognised
- So is a tool for describing specifications – namely a grammar (LALR + disambiguation rules)
- Nowadays substituted by a more powerful tool called bison

Front-end structure



YACC

- Yacc calls a function to obtain the input
  (a lexical analyser)
  Lexer recognises the *terminal symbols = token*
- Control the input process
- Organise tokens wrt the given structure.
  The parser recognises a structure which corresponds to a series of non terminal symbols
- To each grammar production can be bound an action

Basic structure of yacc source file

- Similar to Lex, 3 sections: declarations – grammar + actions – program/user routine

  Declarations
  %%
  rules
  %%
  programs

- As well as lex we can have empty sections but not the rules one

  %% declaration is empty
  rules program is empty

Basic input

- Take the basic file
  (directory 2.0 introduction to yacc - basic)
- This is a senseless grammar invented for approaching purposes.
- Compile it with yacc basic.y
- What did happen?

Basic input (cont.)

- Yacc is enough friendly
- It tell us if the grammar is ambiguous
- Tells us if we have unused production
- Helps us in spotting grammar conflicts..
- We will see some complex examples in which it will prompt very useful messages, concerning the usage of the rules and the actions.
- Try with basic1.y and basic2.y

Let's see the source file: Rules section

- One or more grammar rule
  A: Body1;
  A: Body2;
  A: Body3;

  they can be grouped in
  A: Body1 |Body2 |Body3
- Case sensitive
- All C escapes recognized eg: \n, \r, \f, octal numbers
- Starting symbol specified or by default is the first found with the use of %start symbol
- Let's have an example (firstgrammar.y)

Rules section - Actions

- An action is an arbitrary C statement that can be bound to one or more grammar rule.

EG:

A : (expr) {printf("helloooo");}

- Return or obtain values
- Actions are not executed right immediately, they are based and make use of a stack.
- "$" is used to signal to Yacc the stack position

The lexical analyser..

- Is a function called ***yylex***
- Such function returns a token number corresponding to the token read
- If the token has a value associated it should be associated with a value called ***yylval***
- Does such functions recall you something…?

The lexical analyser..

- Is a function called ***yylex***
- Such function returns a token number corresponding to the token read
- If the token has a value associated it should be associated with a value called ***yylval***
- does such functions recall you something…?
- Lex is designed to work in harmony with Yacc
- Btw Lex is not compulsory – but is really useful

The parser produced

- The parser is just another C program
- The parser is a finite state machine
- It uses a stack
- Can read another input token (***lookahead*** token)
- 4 action possible for such machine
  SHIFT
  REDUCE
  ACCEPT
  ERROR

The parser – parsing rules

- Parsing is done following some intuitive rules.. That are?

The parser – parsing rules

- Parsing is done by following two easy rules
- Look at current state
  if current state is not enough call yylex() and obtain the next token (lookahead)
- Use current state + lookahead
  decide next action and executes it.
  Action may push states on the stack or pop them off the stack.
  (Using a production will consume some tokens).
  Lookahead may be processed or left unconsumed for next interaction.

The parser – accept/error

- Accept → ALL the input was read, the structure of the input fits the specifications provided. The last token read must be the end marker (**$**).
- If we reach such a point the parser did correctly its job.
- Error → some input was read but… the parser can not find a matching respecting the specification given.
- Token so far + lookahead token does not correspond to any legal input (eg. the action in the parsing table is empty)

The parser - shift

- The most common action that the parser undertakes.
- Is used mostly together with the lookahead token which, after the shift action, is cleared
- It tells the parser to move its current state to another

  EG lookahead for IF statements

The parser - reduce

- Happens when right hand of a production is recognized

  EG:

- Replace right hand side with left hand side
- Pop off states from the stack and push a new state on top of it
- Reduce actions manages also the value stack (the one that holds the $ values used in the actions)

The parser - description

- Yacc is really powerful and can produce a file which describe the parser for us (human readable)
- Helpful to manage and correct conflicts
  EG reduce-reduce
  shift-reduce
- Can produce some statistic about the grammar
- Yacc –v file.y
  will produce y.output

Description example

- Let's proceed with an example

```
%token DING DONG DELL
%%
rhyme : sound place;
sound: DING DONG;
place: DELL;
```

- Yacc –v file.y –o yourname
  Open yourname and take a look

  related file is parser_analysis under directory "2.0 - introduction to yacc"

Description example (cont)

- Let's look at the states 0 to 7
- _ or . Used to indicate what has been read so far (recall we are using item sets)
- Suppose input is DING DONG DELL
- Let's proceed in understanding what's the parser job
- The initial state is zero thus we start by looking at that one.

- Look at input in order to decide between actions in state 0
- (state 0) First token in: DING, is read – becoming the lookahead token
  in state 0 upon reading DING we have a shift and ***go to*** state 1
  State 1 is pushed into the stack, lookahead is cleared.
- (state 1) DONG is read becoming the lookahead.
  On token DONG in state 1 action is a shift, and a go to state 4.
  So state 4 is pushed onto the stack and lookahead is cleared.
- (state 4) on the stack we have states 0 1 4
  In state 4 we don't read any token, the default action is a reduce by rule 2 (which is written right above the $default)
  reduce by "sound : DING DONG"
  2 symbols on the right hand side thus 2 states popped of the stack → state 0 is now again on top
- (state 0) for *sound* we have a go to state 3
- (state 3) read DELL, which becomes the lookahead token.
  Action for DELL corresponds to a SHIFT and go to state 6, thus state 6 is pushed onto the stack.

- Recall last action of previous slide
  (state 3) read DELL, which becomes the lookahead token.
  Action for DELL corresponds to a SHIFT and go to state 6, thus state 6 is pushed onto the stack.

- (state 6) like we did for state 4. the default action is undertaken and we do a REDUCE action using rule 3
  reduce using "place : DELL"
  state 6 which is on top of the stack is popped off – due to only having 1 symbol on the right hand side of the production 3

- (state 3) again on state 3, we have for *place (the left hand side of production 3) the action* go to state 7
  the stack contains 0 3 and 7 now, which is the current state

- *(state 7) we reduce by using the rule number 1*
  reduce by "rhyme: sound place"
  we pop off 2 states from the stack (corresponding to the 2 symbols sound and place) state 0 is now uncovered

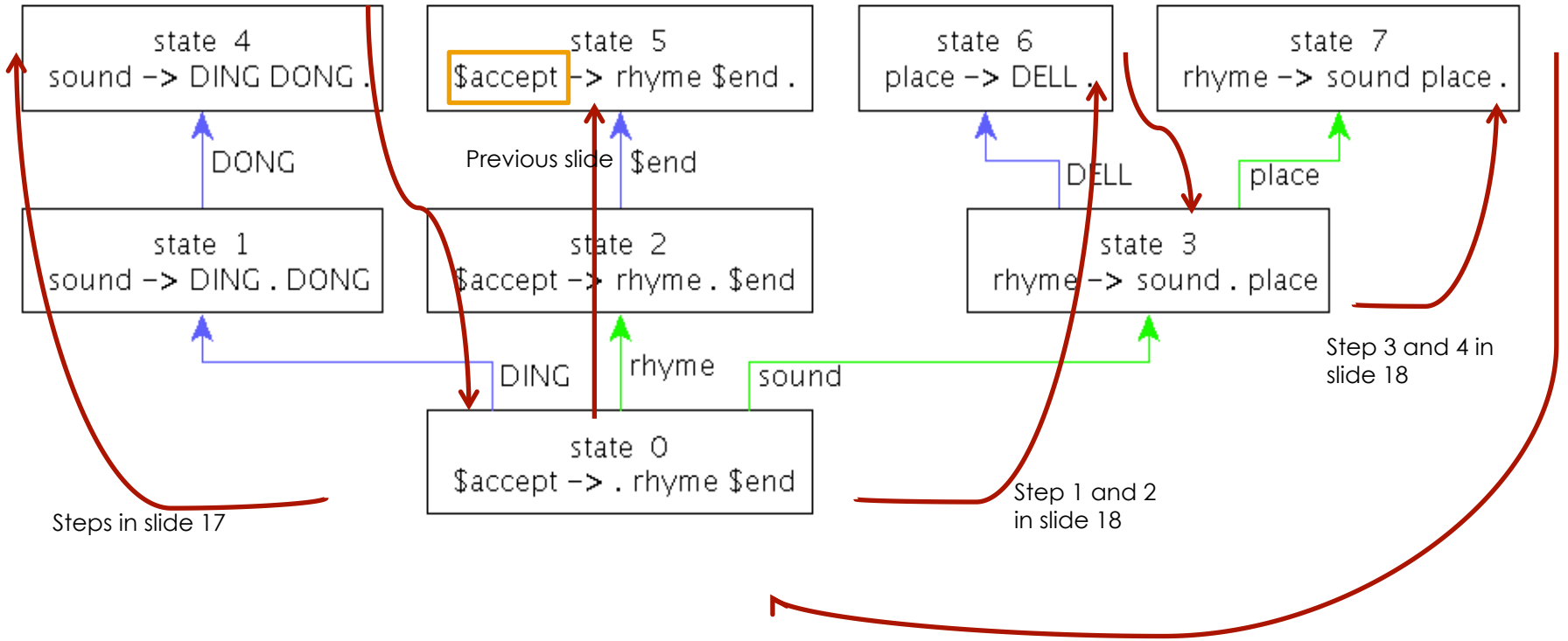
- *(state 0) in state 0, for the LH of production 1 we have a goto 2*

- Recall last action of the previous slide
  (state 0) in state 0, for the LH of production 1 we have a goto 2

- (state 2) – currently on the stack we have 0 and 2
  we read the next token, which in this case signals the end of the inpu → $end
  for such token we have a shift, such lookahead is cleared and we go to state 5 by pushing it onto the stack.

- (state 5) a default action which is the accept one.
  The parser has correctly done it's job.

Parser graph



exercises

Try to proceed as we did in previous slides with the following inputs:

DING DONG DONG

DING DONG

DING DONG DELL DELL

what does happen?

Ambiguous grammars

- What does mean that a grammar is ambiguous?
- Take expr : expr – expr (take – a as token)
  then
  expr – expr – expr
  means..?

Ambiguous grammars

- What does mean that a grammar is ambiguous?
- Take expr : expr – expr (take – a as token)
  then
  expr – expr – expr
  means..?
  (expr - expr) – expr left association
  or
  expr – (expr – expr) right association
- Yacc may find some difficulties in doing a choice when in such a situation

Ambiguous grammar

- Expr – expr – expr
- The parser reads "expr - expr"
  what to do now?
- **REDUCE**… "-expr" remains in the input → another reduce is done
- **DEFER** the application of the rule → read the input until
  expr – expr – expr
- Apply the rule to the rightmost symbols → obtain expr - ***expr***
- Now 2 choises: shift or reduce → the parser doesn't know what to do.

Ambiguous grammar

- Ambiguous grammar are driver for 2 kind of problems related to parsers
- **REDUCE / REDUCE conflict**
  when the parser has a choice of two legal reduction, but doesn't know what to do
- **SHIFT / REDUCE conflict**
  as the previous example, the parser has a choice to shift or reduce, but doesn't know how to act
- Take a look at ambiguous_grammar.y with yacc –v option → first line says
  "State 5 conflicts: 1 shift/reduce"

Disambiguating rules

- Yacc's disambiguating default rules:

  1) in a shift/reduce conflict → shift
  (normally)
  (in certain cases can reduce by default)
  2)in a reduce/reduce conflict → reduce by the earlier grammar rule
  (following the input sequence of specifications)

- An example follows

```
Stat    :       IF '(' cond ')' stat
        |       IF '(' cond ')' stat ELSE stat
        ;
```

INPUT: "IF (C1) IF (C2) S1 ELSE S2"

POSSIBLE STRUCTURE given the previous rules:

```
IF (C1) {
   IF (C2) S1
}
ELSE S2
```

```
IF (C1) {
   IF (C2) S1
   ELSE S2
}
```

This is a clear shift/reduce conflict

The possible and desired grouping (the one on the right) is obtained by applying the disambiguating rule number 1 → shifting and binding the else to the previous if

Precedence of operators

- Resolving conflict is not always sufficient
- Arithmetic expression can not be resolved by removing the conflicts
- We need ***precedence*** of operators and information about left or right **associativity**

A note

- It turns out that
  "ambiguous grammar with appropriate disambiguating rules can create parsers that are faster and easier to write than those from unambiguous grammars"
- can anyone guess why?

Yacc – precedence

- Precedence management is attached to tokens (in the declaration section - obviously)

- %left '+' '-'
- %left '*' '/'
- Lines must be listed in order of increasing precedence (from botton to top)
- See examples disambiguating1 disambiguating2

Yacc – precedence (cont)

- When precedence mechanism is used, unary operators must be given a precedence
- Take '–'
  is meant as binary operator
  is meant as unary operator
  How to manage such situation?
- Use %prec in the rules, right after the body – followed by the token or literaly- before semicolon.
  EG:

```
expr :  ....
      | ....
        '-' expr %prec '*'
      ...
```

Disambiguating rules – part 2

- Associativity and precedence raise new disambiguating rules

1) **precedence** and **associativity** are recorded for those tokens and literals that have them declared

2) A prec. or association is bound with each rule: it is the prec or association of the last token or literal in the body.
   If **%prec** is found it overrides the previous rule (previous sentence)

3) If reduce/reduce or shift/reduce is found and no prec or association is declared → apply previous rules (see disambiguating rules slide)

4) If shift/reduce is found and both the grammar rule and the input char have precedence and associativity associated with them → conflict solved in favour of action with higher precedence.

Yacc environment

- Yacc turns specifications into a C program
- Yacc generates a function called ***yyparse()*** which calls repeatedly yylex
- Main must be defined if we want to manage some stuff (including calling yyparse).
- Yyerror is called when an error is found, thus we can write this function as well to provide custom error management
- Yacc has a library with a default version of main and yyerror

Conclusion

- Yacc provides a parsing tool
- It needs an input tokenised stream to organize the structure
- Lex provides a way to tokenise an input stream
- Next time we will put together lex and yacc

Exercises

- For next time try the following exercises
- Try to parse the following grammars with Yacc, decide which are the tokens and which the LH symbols. If you find conflicts try to fix them by using disambiguating rules
- S → AB;
  A → a | ε ;
  B → bB | b;
- S → a | AbC;
  A → a;
  C → A | c;

Exercises

- Statement → IF '(' Boolexpr ')' Statement | IF '(' Boolexpr ')' Statement ELSE Statement | Expr ;

  Boolexpr: bool ;

  Expr : id ;

  NOTE This is really simple exercise for the sake of playing with yacc is ok. Statements have much more declaration as well as boolexpr and expr. Is just an exercise.

Exercises

- phrase → cart_animal AND CART
  | work_animal AND PLOW;

  cart_animal → HORSE | GOAT;
  work_animal → HORSE | OX;

- Is the grammar ambiguous? Is the grammar LALR? LR (0) or SLR?
- Think about a grammar for a simple calculator.

Bibliography

- Yacc – yet another compiler compiler
- Lex and yacc tutorial by Tom Niemann
- Lex and yacc O'reilly
- http://docs.oracle.com/cd/E19620-01/805-4035/6j3r2rqn3/index.html