Yacc Parser and Conflicts

CS 315 – Programming Languages

Pinar Duygulu

Bilkent University

How the parser works

- Yacc turns the specification file into a C program which parses the input according to the specifications given
- The parser produced by yacc consists of a finite state machine with a stack
- The parser is capable of reading and remembering the next input token (lookahead token)
- The current state is always the one on top of the stack
- Initially the stack contains initial state (state 0) and no lookahead token has been read

How the parser works

- The machine has only for actions : shift, reduce, accept, error
- Based on the current state, the parser decides whether it needs a lookahead token to decide what action should be done, if it needs one and does not have one it calls yylex to obtain the next token
- Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may results in states being pushed onto the stack, or popped off the stack, and in the lookahead token being processed or left alone

Shift action

<lookahead_token> shift <state>

e.g.

```
IF shift 34
```

When lookahead token is IF push down the current state on the stack, put state 34 onto stack and make it current state, clear the lookahead symbol

Reduce action

When the parser has seen the right hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule, replace the right hand side by left hand side

e.g:

```
. reduce 18
```

means reduce grammar rule 18

e.g.

```
A : x y z;
```

pop off the top three states (number of rules on the RHS) from the stack, then perform

```
A goto 20
```

causing state 20 to be pushed onto stack, and become the current state

accept and error

- Accept action indicates that the entire input has been seen and that it matches the specifications
- Appears only when the lookahead symbol is the endmarker and indicates that the parser has successfully done its job

- The error action represents a place where the parser can no longer continue parsing according to the specifcation. The input tokens it has seen together with the lookahead token cannot be followed by anything that would result in a legal input

example

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

```
$yacc -v filename.y
```

produces a file named y.output
it is a human readable description of the parser

example

```
state 0
    $accept: _rhyme  $end
    DING  shift 3
    .  error
    rhyme  goto 1
    sound  goto 2

state 1
    $accept:rhyme_$end
    $end  accept
    .  error

state 2
    rhyme:sound_place
    DELL  shift 5
    .  error
    place   goto 4

state 3
    sound:DING_DONG
    DONG shift 6
    .  error

state 4
    rhyme:sound  place_  (1)
    .   reduce  1

state 5
    place:DELL_     (3)
    .   reduce  3

state 6
sound : DING  DONG_     (2)
.   reduce  2
```

Example (for input DING DONG DELL)

```
state 0
    $accept:  _rhyme  $end
    DING  shift 3
    .  error
    rhyme  goto 1
    sound  goto 2
```

- Initially current state is 0
- First token DING is read
- Action in state 0, on token DING is shift 3
- Push state 3 onto stack
- Clear the lookahead symbol
- Make state 3 current state

- Initial stack : 0
- Current stack : 0 3

Example (for input DING DONG DELL)

```
state 3
    sound:DING_DONG
    DONG shift 6
    .   error
```

- Read the next token: DONG
- It becomes the lookahead symbol
- The action in state 3 on token DONG is shift 6
- Push state 6 onto stack
- Clear lookahead token

- Current stack : 0 3 6

Example (for input DING DONG DELL)

```
state 6
sound : DING  DONG_     (2)
.   reduce  2
```

- In state 6 without even consulting the lookahead the parser reduces by rule 2
- sound : DING DONG
- This rule has two symbols on the right hand side,
- so two states, 6 and 3, are popped off of the stack, uncovering state 0

- Current stack : 0

Example (for input DING DONG DELL)

```
state 0
    $accept:  _rhyme  $end
    DING  shift 3
    .  error
    rhyme  goto 1
    sound  goto 2
```

- In state 0, look for a goto on sound
- Push state 2 onto stack
- State 2 becomes current state

- Current stack : 0 2

Example (for input DING DONG DELL)

```
state 2
    rhyme:sound_place
    DELL  shift 5
    .  error
    place   goto 4
```

- Next token is DELL
- The action in state 2 on token DELL is shift 5
- Push state 5 onto stack
- Make it current state
- Clear lookahead symbol

- Current stack : 0 2 5

Example (for input DING DONG DELL)

```
state 5
    place:DELL_      (3)
    .    reduce  3
```

- In state 5 the only action is reduce by rule 3
- It has one symbol in the right
- So one state (state 5) is popped from the stack, and state 2 is uncovered

- Current stack : 0 2

Example (for input DING DONG DELL)

```
state 2
    rhyme:sound_place
    DELL  shift 5
    .  error
    place   goto 4
```

- In state 2, goto on place results in state 4
- Push state 4 onto stack

- Current stack : 0 2 4

Example (for input DING DONG DELL)

```
state 4
    rhyme:sound  place_  (1)
    .    reduce  1
```

- In state 4 only action is reduce 1
- There are two symbols on the right
- Pop off two states from the stack
- Uncover state 0

- Current stack : 0

Example (for input DING DONG DELL)

```
state 0
    $accept:  _rhyme  $end
    DING  shift 3
    .  error
    rhyme  goto 1
    sound  goto 2
```

- In state 0, goto on rhyme causes the parser to enter state 1
- Push state 1 onto stack
- Make state 1 current state

- Current stack : 0 1

Example (for input DING DONG DELL)

```
state 1
    $accept:rhyme_$end
    $end  accept
    .  error
```

- In state 1 the input is read and endmarker is obtained ($end)
- The action is accept
- Successfully end the parser

Pointer model

- A pointer moves (right) on the RHS of a rule while input tokens and variables are processed

```
% token A B C
%%
start : _A B C
/* after reading A:
 start : A_B C */
```

- when all elements on the RHS are processed (pointer reaches the end of a rule) the rule is reduced
- If a rule reduces, the pointer returs to the rule it was called

Conflicts

- There is a conflict if a rule is reduced when there is more than one pointer
- Yacc looks one token ahead to see if the number of tokens reduces to one before declaring a conflict
- Example:

```
% token A B C D E F
%%
start : x | y
x: A B _ C D;
y: A B _ E F ;
```

- After tokens A and B, either one of the tokens, or both will disappear.
- If the next token is E the first, if the next token is C the second will disappear
- If the next token is anything other than C or E both will disappear – no conflict

Conflicts

- The other way for pointers to disappear is for them to merge in a common subrule

Example:

```
%token A B C D E F
%%
start : x| y
x: A B _ z E;
y: A B _ z F;
z : C D;
```

Conflicts

- Initially there are two pointers. After reading A and B these two pointers remain.
- Then these two pointers merge in the z rule.
- The state after reading token C is :

```
%token A B C D E F
%%
start : x| y
x: A B z E;
y: A B z F;
z : C _ D;
```

Conflicts

- However after reading A B C D this pointer splits again into two pointers

```
%token A B C D E F
%%
start : x| y
x: A B z _ E;
y: A B z _ F;
z : C D;
```

- Note that yacc looks one token ahead before declaring any conflict. Since one of the pointers will disappear depending on the next token. Yacc does not declare any conflict

Conflicts

- Conflict example

```
%token A B
%%
start : x B | y B ;
x : A ;   reduce
y : A ;   reduce
```

- reduce/ reduce conflict on B

- After A there are two pointers. Both rules x and y want to reduce at the same time.
- If the next token is B, there will be still two pointers.
- Such conflicts are called reduce/reduce conflict

Conflicts

- Another type of conflict occurs when one rule reduces while the other shifts. Such conflicts are called shift/reduce conflicts

```
%token A R
%%
start : x | y R ;
x : A _ R; shift
y : A ; _  reduce
```

- shift/reduce conflict on R
- after A, y rule reduces x rule shifts. The next token for both cases is R

Conflicts

```
%token A
%%
start : x | y ;
x : A ;  _ reduce
y : A ;  _  reduce
```

- reduce/reduce conflict on $end
- At the end of each string there is a $end token. Therefore yacc declares reduce/reduce conflict on $end

Conflicts

- Empty rules
- ```
  %token A B
  %%
  start : empty A A
        | A B;
  empty : ;
  ```

- Without any tokens
- ```
  %token A B
  %%
  start : empty A A
        | _ A B;
  empty : _ ;
  ```

  shift/reduce conflict on A

  If the next token is A the empty rule will reduce and second rule (of start) will shift. Therefore yacc declares shift/reduce conflict on A