EECS483 D5: Project 2 Details

Chun-Hung Hsiao

Feb 8, 2013

Announcements

- Grades of Project 1 and Homework 1 have been released on CTools
  - Contact me if there is any problem
- Homework 2 is due on Monday, Feb 11
  - Hand it in class with the cover page
- Project 2 is due on Wednesday, Feb 13

Yacc Syntax

- Structure of a .y file:

  DECLARATIONS
  %%
  RULES
  %%
  USERCODE

Connecting Lex and Yacc

- Use `%token` in DECLARATIONS to define terminals

```
%token T_Number
```

- Use `yacc -h` to generate `y.tab.h`

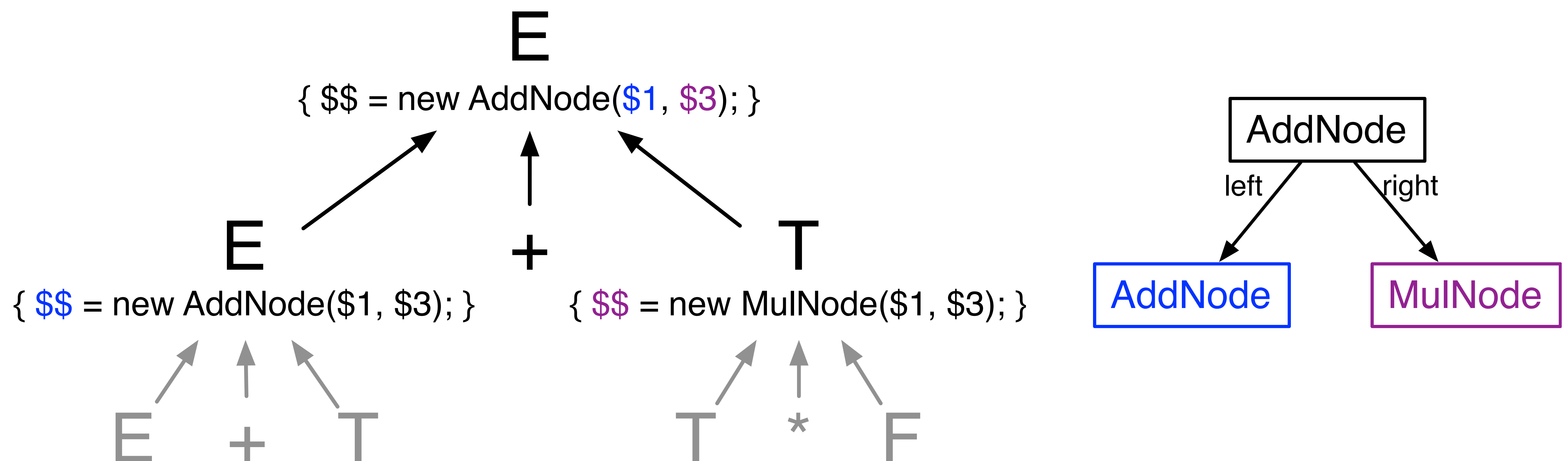
```
enum yytokentype {
        T_Number = 258
};
```

- Include `y.tab.h` in `scanner.l` and return the corresponding terminal for each pattern

```
[0-9]+   return T_Number;
```

Constructing AST

- Yacc uses bottom-up parsing (LALR(1))
- An action is performed when a handle is reduced through its corresponding rule
  - At the time the handle is reduced, all actions associated with its RHS are already performed



Semantic Value Types (1/2)

- `$$` and `$`*n* refer to `yylval` of LHS and RHS
- Use `%union` to define fields in `yylval`

```
%union {
        int number;
        ExprNode *expr;
}
```

- It generates the union type for `yylval` in `y.tab.h`

```
typedef union YYSTYPE {
        int number;
        ExprNode *expr;
} YYSTYPE;
extern YYSTYPE yylval;
```

Semantic Value Types (2/2)

- For a terminal, specify its `yylval` field with `%token`

```
%token <number> T_Number
```

- For a nonterminal, specify its field with `%type`

```
%type <expr> E T F
```

- Then the following action

```
E : E + id { $$ = new AddNode($1, $3); }
```

becomes

{ yylval^{LHS}.expr = new AddNode(
    yylval^{RHS1}.expr, yylval^{RHS3}.number); }

Ambiguous Grammar : Operator Precedence & Associativity (1/2)

- Consider the following grammar

```
E : E '+' E | E '-' E | E '^' E
  | E '=' E | T_Number ;
```

- This grammar is ambiguous
  - Why?
- The ambiguity can be resolved with `%left`, `%right` and `%nonassoc`

```
%nonassoc '='
%left '+' '-'
%right '^'
```

Ambiguous Grammar : Operator Precedence & Associativity (2/2)

- How about the following grammar

```
E : E '-' E | '-' E | T_Number ;
```

- The ambiguity can be resolved by adding a dummy token to specify the precedence of the negation

```
%left '-'
%left NEG
%%
E : E '-' E
  | '-' E %prec NEG
  | T_Number
  ;
```

Ambigious Grammar: Dangling Else

- Consider the following grammar

```
Stmt : Expr
     | if Expr then Stmt
     | if Expr then Stmt else Stmt
     ;
```

- This grammar is ambiguous
  - Why?

Dangling Else: Solution I

- Redesign the language specification to avoid it!
  - `try-catch` in C++
  - `endif` in many languages
- Unfortunately we cannot do this in PP2...

Dangling Else: Solution II (1/2)

- Assign new semantic: pair each `else` with closest `if`
  - `MatchedStmt`: each `if` is paired with an `else`
  - `UnmatchedStmt`: contains `if` without a matched `else`

- Rewrite the grammar

```
Stmt : MatchedStmt | UnmatchedStmt ;
MatchedStmt
   : if Expr then _______ else _______
   | Expr ;
UnmatchedStmt
   : if Expr then _______ else _______
   | if Expr then _______ ;
```

Dangling Else: Solution II (1/2)

- Assign new semantic: pair each `else` with closest `if`
  - `MatchedStmt`: each `if` is paired with an `else`
  - `UnmatchedStmt`: contains `if` without a matched `else`
- Rewrite the grammar

```
Stmt : MatchedStmt | UnmatchedStmt ;
MatchedStmt
   : if Expr then MatchedStmt else ______
   | Expr ;
UnmatchedStmt
   : if Expr then ______ else ______
   | if Expr then ______ ;
```

Dangling Else: Solution II (1/2)

- Assign new semantic: pair each `else` with closest `if`
  - `MatchedStmt`: each `if` is paired with an `else`
  - `UnmatchedStmt`: contains `if` without a matched `else`
- Rewrite the grammar

```
Stmt : MatchedStmt | UnmatchedStmt ;
MatchedStmt
   : if Expr then MatchedStmt else MatchedStmt
   | Expr ;
UnmatchedStmt
   : if Expr then _______ else _______
   | if Expr then _______ ;
```

Dangling Else: Solution II (1/2)

- Assign new semantic: pair each `else` with closest `if`
  - `MatchedStmt`: each `if` is paired with an `else`
  - `UnmatchedStmt`: contains `if` without a matched `else`
- Rewrite the grammar

```
Stmt : MatchedStmt | UnmatchedStmt ;
MatchedStmt
   : if Expr then MatchedStmt else MatchedStmt
   | Expr ;
UnmatchedStmt
   : if Expr then MatchedStmt else ______
   | if Expr then ______ ;
```

Dangling Else: Solution II (1/2)

- Assign new semantic: pair each `else` with closest `if`
  - `MatchedStmt`: each `if` is paired with an `else`
  - `UnmatchedStmt`: contains `if` without a matched `else`
- Rewrite the grammar

```
Stmt : MatchedStmt | UnmatchedStmt ;
MatchedStmt
   : if Expr then MatchedStmt else MatchedStmt
   | Expr ;
UnmatchedStmt
   : if Expr then MatchedStmt else UnmatchedStmt
   | if Expr then ______ ;
```

Dangling Else: Solution II (1/2)

- Assign new semantic: pair each `else` with closest `if`
  - `MatchedStmt`: each `if` is paired with an `else`
  - `UnmatchedStmt`: contains `if` without a matched `else`
- Rewrite the grammar

```
Stmt : MatchedStmt | UnmatchedStmt ;
MatchedStmt
   : if Expr then MatchedStmt else MatchedStmt
   | Expr ;
UnmatchedStmt
   : if Expr then MatchedStmt else UnmatchedStmt
   | if Expr then   Stmt   ;
```

Dangling Else: Solution II (2/2)

- How about the following grammar?

```
Stmt : Expr
     | if Expr then Stmt
     | if Expr then Stmt else Stmt
     | while Expr Stmt
     ;
```

Dangling Else: Solution III

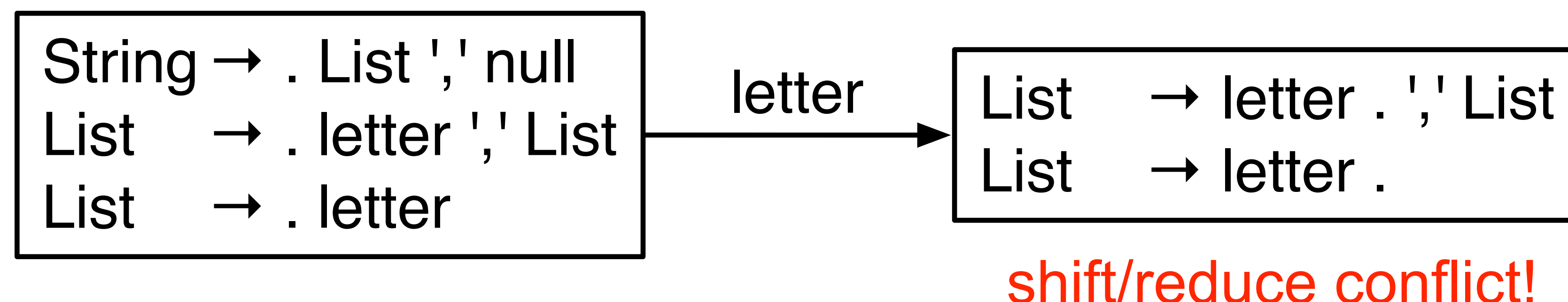
- Use the same trick as we do for operator precedence!
  - `%left`, `%right`, or `%nonassoc`
  - `%prec`

Unambiguous Grammar w/ Conflicts (1/2)

- Consider the following grammar

```
String : List ',' null ;
List   : letter ',' List
       | letter ;
```

- Grammar is unambiguous but contains a shift/ reduce conflict
  - Both lookahead sets of `letter` and `List` contain `','` so cannot decide whether to shift or reduce



Unambiguous Grammar w/ Conflicts (2/2)

- Idea: Make the lookahead sets disjoint!
- Solution 1

```
String : List null ;
List   : letter ',' List
       | letter ',' ;
```

- Solution 2

```
String : List ',' null ;
List   : List ',' letter
       | letter ;
```

More about Conflicts

- Shift/reduce conflicts in Bison
  - http://www.gnu.org/software/bison/manual/html_node/Shift_002fReduce.html#Shift_002fReduce
- Reduce/reduce conflicts in Bison
  - http://www.gnu.org/software/bison/manual/html_node/Reduce_002fReduce.html#Reduce_002fReduce

Classifying Grammars & Languages

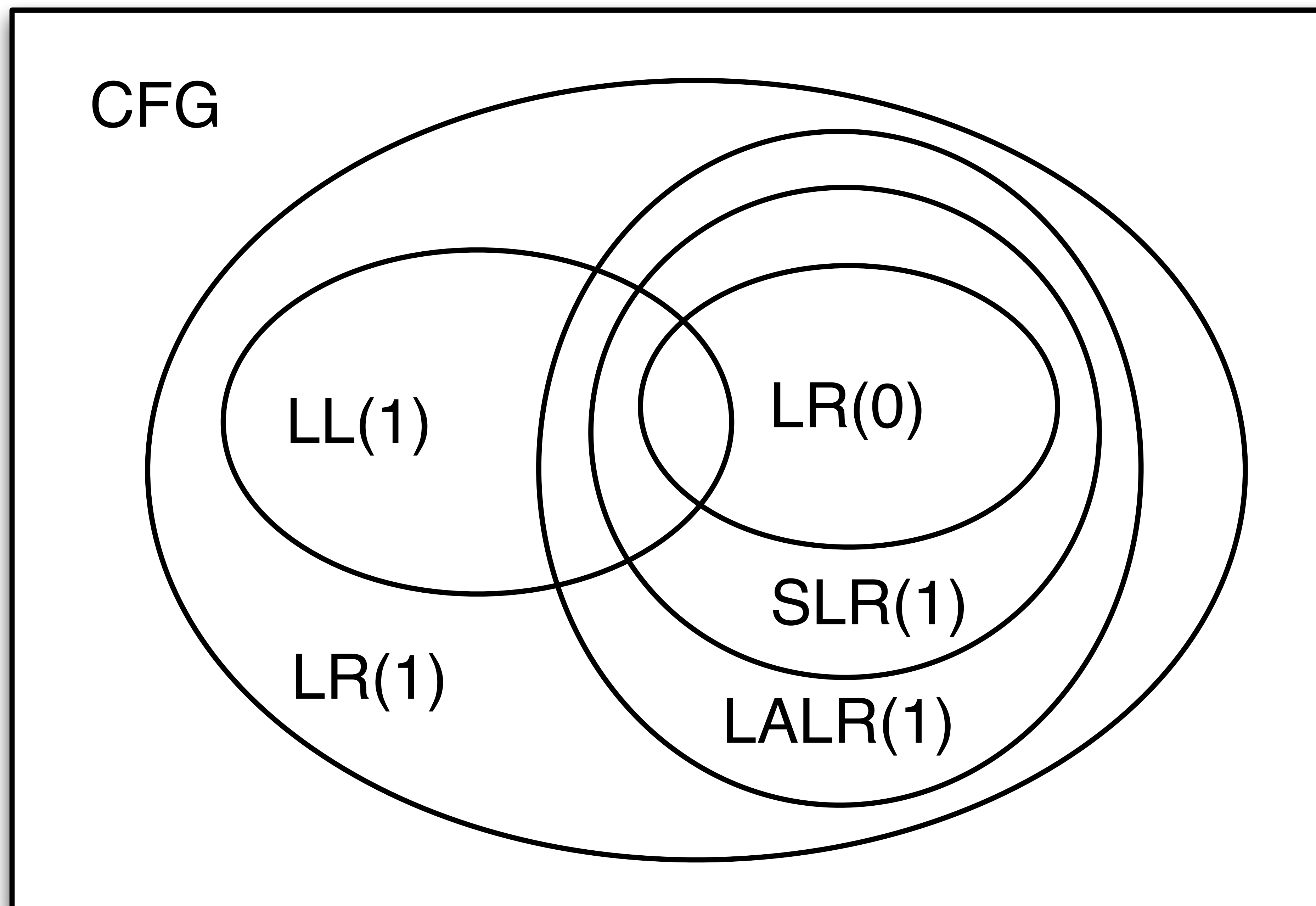
- Now we have many different parsing algorithms
  - LL(1), LL(k), LR(0), LR(1), SLR(1), LALR(1), LR(k), ...
- These parsing algorithms can be used to classify grammars and languages
- A grammar *G* is in class *C* if it can be parsed with algorithm *C* without any conflict
- A language *L* is in class *C* if there exists a class-*C* grammar that recognizes *L*

The Power of LR(1)

- Any LR(0) grammar is LR(1).
- Any LL(1) grammar is LR(1).
- Any deterministic CFL (a CFL parseable by a *deterministic pushdown automaton*) has an LR(1) grammar.
- Any LL(k) *language* is LR(1), though individual LL(k) *grammars* might not be.
- Any LR(k) *language* is LR(1), though individual LR(k) *grammars* might not be.

http://www.eecs.umich.edu/courses/eecs483/w13/w13_lectures/lecture-6.pdf, p.385

The Classes of Grammars



Thanks & Happy Asian New Year!