03/17/2010

# Recitation #8

## Administrative

- Assignments 3 is graded
  - see grading criteria and reference solution on my Wiki page
  - Ambiguity was not graded
  - Conclusion of what the table result means was expected

## CKY – Recap

- **Ambiguity**
  - Sometimes there is an ambiguity concerning which production rule to add to the table
  - When Dr. Hughes asks you about CKY in the next exam, you should know how to handle it
  - Example (from assignment 3):

    **S → AB | BA** **A → CD | a** **B → CE | b**
    **C → a | b** **D → AC** **E → BC**

| | a | b | b | a |
|---|---|---|---|---|
| 1 | A → a<br>C → a | B → b<br>C → b | B → b<br>C → b | A → a<br>C → a |
| 2 | S → AB<br>D → AC | E → BC | S → BA<br>E → BC | |
| 3 | B → CE | B → CE | | |
| 4 | S → AB<br>S → BA<br>E → BC | | | |

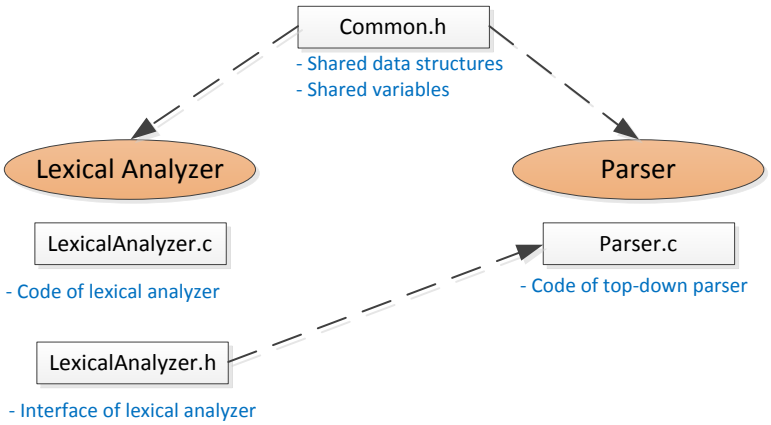
  - 1st (leftmost) derivation: **S** → AB → aB → aCE → abE → abBC → abbC → **abba**
  - 2nd (leftmost) derivation: **S** → BA → CEA → aEA → aBCA → abCA → abbA → **abba**

- **Construct parse tree**
  - To do that we need to keep track on which table entries we combined when we add a non-terminal to a table cell
  - At the end, reconstruct parse from the back-pointers I kept
  - Illustrate the concept with the graphical representation of Exorciser

## Assignment 4

- **Make sure to install Bison in path without spaces**
- Dr. Hughes uploaded a new specification document that provides full explanation of the assignment

- Recommended general structure of the code:




Look at actual code files

- Remember generic recursive descent parser you had to write for exam #1:

*case_stmt* ::= **CASE** *expression* **OF** *case_element* { '**;**' *case_element* } **END**

and its associated top-down parsing pseudo-code:

```
void caseStatement( )
{
    getNextToken();
    expression();
    if (SY == OF)
    {
        getNextToken();
        caseElement( );
        while (SY == SEMICOLON)
        {
            getNextToken( );
            caseElement();
        }
        if (SY == END)
            getNextToken( );
        else error( );
    }
    else error( );
}
```

That is exactly how your code should work, i.e. there should be a function for each non-terminal in the grammar.

### *Lexical Analyzer Code*

- You can choose how to implement the lexical analyzer
  - Use your own code from assignment 1
  - Use Dr. Hughes reference solution for assignment 1
  - Use code from your class mates (proper credit to author)
  - Use Flex-generated scanner


- Lexical analyzer is not monolithic code that processes the whole input file, but should return token by token → Don't go the detour of letting the lexical analyzer write the tokens to a separate output file and then read that from the parser!

### *Specific Areas of Interest*

- **Intermediate Code**
  - Consists of triples (opcode, operand, operand)
  - See `emit()` function; is called for each generated triple
  - See table of opcodes and their meaning in Dr. Hughes' description
  - Print out code after each line of parsed code with function `dumpcode()` (see line 55 in While.l and lines 305ff of While.y)
  - Print out all the generated code at the end of processing

- **Errors / Warnings**
  - Lexical analyzer doesn't need to handle errors beyond printing "unrecognizable character" and returning the BAD token (lines 56-57 in While.l)
  - See `error()` and `warning()` functions in While.y (lines 328 – 337)
  - Errors in parser are usually triggered if grammar rules are violated
  - Error recovery:
    - See ok() function
    - It will search for a synchronizing token (WHILE, SEMICOLON, and custom) after an error occurred
    - See example in var.list and statement
  - Only warning is for duplicate variable declarations (lines 93 – 97)

- **Symbol Table**
  - If identifier (alphanumeric and starts with alphabetic character) is recognized by scanner, call `setid()` → copy name into 0-th element of symbol table, linear search for symbol, `strcmp()` will return 0 if found, so `yylval` will be > 0 if duplicate symbol; otherwise `yylval` will be set to 0
  - `yylval` is then checked in lines 93 and 96 (as $1); if the variable name is not already in symbol table call `enter()` function
  - Keeps track of read/write references to variables (see `emit()` function)

- **Number handling**
  - All variables are integers → no explicit typing required
  - If number is recognized by lexical analyzer, convert to integer in `setnum()` function and store value in `yylval`

- **Arithmetic Operations**
  - Keeps track of result of intermediate operation by saving the location of the generated code with `$$ = -triple` → See generated code from 'good_program.txt' as example
  - Precedence / Associativity is not handled by grammar rules (the way we learned it), but by Bison specification:

    ```
    %left PLUS MINUS
    %left TIMES DIVIDE
    ```

    Mean that PLUS, and MINUS are left-associative and have lower precedence than left-associative TIMES, and DIVIDE tokens

  - The actual expression grammar on lines 200 – 222 of While.y is as follows:
    $exp \rightarrow exp + exp \mid exp - exp \mid exp * exp \mid exp / exp \mid (exp) \mid NUMBER$
  - You cannot use this grammar in your top-down parser for multiple reasons:
    1. For starters, the given grammar doesn't handle associativity or precedence, because Bison is handling that internally. You need to build an expression grammar similar to the ones we did in class / lab, e.g.
       $A_1 \rightarrow A_1 + A_2 \mid A_1 - A_2 \mid A_2$
       $A_2 \rightarrow A_2 * A_3 \mid A_2 / A_3 \mid A_3$
       $A_3 \rightarrow (A_1) \mid NUMBER$

    2. In addition, you will have to remove the left recursion, because a top-down parser cannot handle left recursion (Bison has no such qualms because it builds a bottom-up parser). Also see pages 265 – 266 of your textbook. In our example, the non-left-recursive version of the grammar is:
       $A_1 \rightarrow A_2 A_1'$
       $A_1' \rightarrow +A_2 A_1' \mid -A_2 A_1' \mid \varepsilon$
       $A_2 \rightarrow A_3 A_2'$
       $A_2' \rightarrow *A_3 A_2' \mid /A_3 A_2' \mid \varepsilon$
       $A_3 \rightarrow (A_1) \mid NUMBER$

- **While / Do constructs**
  - triple (code pointer) is returned on start of while statement (while.start), after whole WHILE construct is parsed ("while" grammar rule), set JUMP point to negative of saved triple → similar handling of DO
  - Backward jump in while is done by `emit("JUMP");forwardTest =1` will jump beyond the end of the while construct if the condition is not fulfilled
  - Backward jump in do is done by setting `forwardTest=0` and backpatching the test

- **Tests (test: rule)**
  - Subtracts 2nd operand from 1st and compares the results
  - Whenever a test is used I need to figure out to which intermediate code triple to jump if the test fails. This destination triple is only available after the whole construct is parsed → use `backpatch()` and `backpatch1()` functions.

- **Embedded Actions**
  - Even though bison's parsing technique allows actions only at the end of a rule, bison can simulate actions embedded within a rule. If you write an action within a rule, bison invents a rule with an empty right-hand side and a made-up name on the left, makes the embedded action into the action for that rule, and replaces the action in the original rule with the made-up name. For example, these are equivalent:

```
thing: A { printf("seen an A"); } B ;
thing: A fakename B ;
fakename: /* empty */ { printf("seen an A"); } ;
```

- Talk about $$, $1, $2 notation in Bison