CSC488S/CSC2107S Compilers and Interpreters **Winter 2014/2015**

Mid Term Test Solution

1. [10 marks]

Leftmost S => ABC => aABC => aBC => aBbC => aBbbC => abbC => abbcC => abbc

Rightmost S => ABC => ABcC => ABc => ABbc => ABbbc => Abbc => aAbbc => abbc

2. [20 marks] The solution below is inspired by a particularly clever student solution.

```
boolean function parseFor( ) {
    return
        getToken()                          // skip over for
        and variable()
        and getToken() and nextToken == ':'         // : and = two tokens
        and getToken() and nextToken == '='
        and expression()
        and getToken()(
        and ( ( nextToken == 'to'                   // to, by case
            and expression()
            and getToken()
            and ( ( nextToken == 'by'
                and expression()
                and getToken()
            or  true ) )
            or ( nextToken == ','                   // list of expressions
                and multiExpression() )  )
        and nextToken == 'do
        and getToken()
        and statements()
        and nextToken == 'end'
        and getToken()
}
boolean function multiExpression() {
    return
        expression()
        and getToken()
        and (  ( nextToken = ','
            and multiExpression()
        or true
}
```

3. [20 marks]

Rule 1 really had to be done during lexical analysis because only the lexical analyzer sees newlines. This rule can be easily implemented with a small amount of buffering in the lexical analyzer. Keep track of the previously emitted token and whether a given line was empty or not. When the lexical analyzer encounters a newline, check whether the previous token was on the Rule 1 list and if it was, emit a semicolon.

Schemes that involved having the lexical analyzer send a newline toke to syntax analysis were not acceptable. They would have made a real mess of parsing.

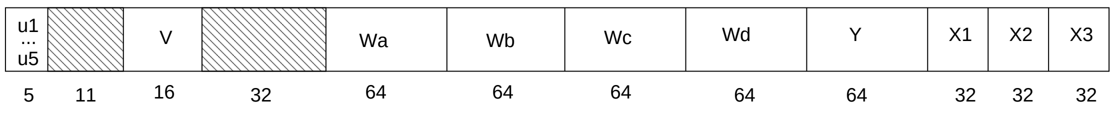
Rule 2 really had to be done during syntax analysis. The lexical analyzer doesn't have enough context to determine if a ")" or "}" is a **closing** bracket or not.

A lot of answers involved changing the grammar to allow statements with or without semicolon in a closing position. These solutions were allowed although they would have caused some serious shift/reduce conflicts.

A (sneaky) solution would be to not change the grammar at all. A missing semicolon before a closing bracket would produce a syntax error. Implement a special case syntax error repair for this error that silently inserted the missing semicolon.

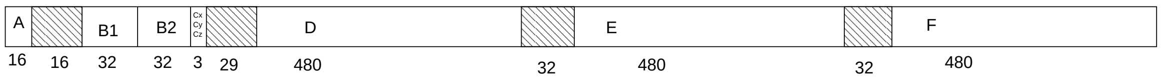
4. [25 marks]

First map the class *inner*



length 480 bits, bit alignment 0 mod 64

Then map the class *outer*



length 1216 bits, bit alignment 0 mod 64

Common problems

- confusing bits and bytes
- adding fill at the end of inner
- missing the fill between D,E and E,F in outer
- wrong fill after Cx,Cy,Cz in outer
- mapping a copy of inner at the beginning of outer

5. [20 marks]

When a **forward** declaration is encountered:

- perform the usual check for duplicate declaration.
- Create a symbol table entry for the function/procedure, record the parameters (if any) and their types. Record the return type of functions.
- Mark the symbol table entry for this declaration as a forward declaration

When a non-forward declaration of a function/procedure is encountered and the name of the function/procedure matches a previous forward declaration:

- For function declarations, check that the forward and actual declarations have the same return type.
- Check that both declarations have the same number of parameters.
- Check that corresponding parameters in the forward and actual declaration have the same type.
- Don't create a new symbol table entry, but unmark the symbol table entry as being forward now that an actual declaration has been found.

At the end is each major scope, check the scope's symbol table for unsatisfied forward declarations and emit an error message if any are found.