

Examination 1 Programming Languages and Systems Concepts,
CSI 511
Fall 2001 (October 24, 2001)

1 Rules of the Exam

This examination is open book and notes. Calculators are permitted. Networked devices are strictly prohibited. The questions are marked as to their relative value, the exam will be scored out of 100% but is worth 25 points towards your course grade. Relax and try to do what you can.

2 The Problem Set

1. Readings (total 15 %) Note: Word for word copying from the readings will not get credit, answer IN YOUR OWN WORDS:

- (a) Give some cases where it is better to use a scripting language than a systems programming language? Do you agree with Ousterhout?(5 %)

Scripting approaches work well where you want higher level functionality or wish to integrate lower level components to get high level functionality. Speed critical, hardware manipulation, and systems software tools tend to be poor candidates for scripting. I tend to agree with Ousterhout, different tools work better for different jobs.

- (b) Meyer's design by contract paradigm focuses on deriving preconditions, postconditions and invariants. Should preconditions be strong or weak? Should postconditions be strong or weak? Why? (5 %)?

Preconditions should be weak, as a weak precondition indicates that the client is less sensitive to its inputs and more robust/easier to use. Postconditions should be strong, since guarantees of output quality are a sign of robustness and again promote ease of use.

- (c) What does Graham think the power of Lisp compares to other languages? Why?(5 %)?

Graham thinks Lisp is the most powerful computing language available, primarily because Lisp enables users to express programs as lists and execute them (the Macro facility).

2. Theoretical Foundations (Automata) (25%)

- (a) Consider the following languages, are they regular, why or why not (10%?)
 - i. $a^i b^i$ where $0 \leq i \leq n$ and n is finite (hint: consider a case where n is small) (5 %)

Yes. Imagine if $n = 1$, we could make an automata that accepts the language ab with a finite number of states. If $n = 2$, we could design an automata which accepts $aabb$ with a finite number of states, and then have a nondeterministic (epsilon or as the book/lecture notes call it, a lambda transition) select between it and the previous automata. By induction, we can see that if we can accept a language $a^k b^k$, where k is finite, (which we can prove by constructive proof) then we can accept $a^{k+1} b^{k+1}$. Using the NFA approach of combining automata using epsilon productions, we can create the automata with a finite (but potentially large) number of states.

ii. $a^n b^n$ when $0 < n$ (5 %)

Since we are given no guarantees that n is finite, the language is not recognizable by a DFA (since an infinite number of states is needed to recognize the language when n is infinite).

(b) Sort the ordering in terms of increasing power (that is their ability to recognize languages): Nondeterministic finite state automata, deterministic push down automata, deterministic finite state automata, non deterministic pushdown automata (5 %).

In order of increasing power (least powerful first), we get:

- Finite State Automata (Deterministic and Nondeterministic are equivalent)
- Deterministic push down automata
- Nondeterministic push down automata

(c) Is it possible to convert the non deterministic finite state automaton shown in Figure 1 into a deterministic finite state automaton? If so, give the corresponding deterministic finite state automaton and the regular expression for the language accepted (10 %).

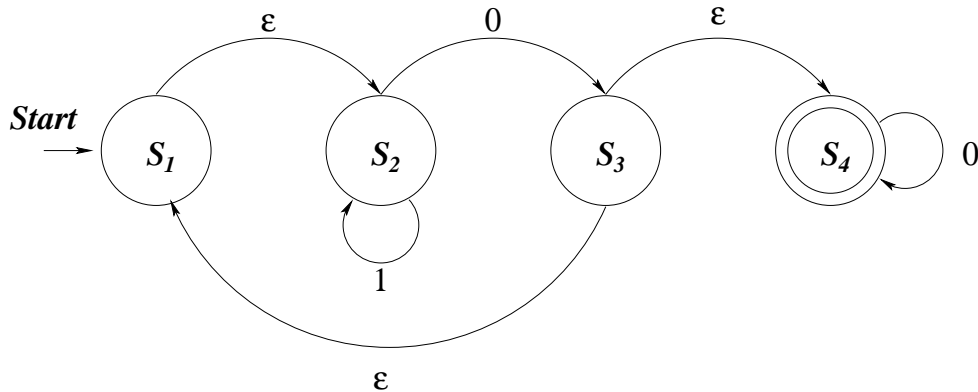


Figure 1: Non Deterministic Finite Automata for Problem 2c

Yes, there is a DFA corresponding to this NFA. When I drew the figure, I accidentally fell back on using the notation ϵ (epsilon) to indicate the empty string, however, the book and lecture notes use λ (lambda). I gave full credit if you assumed that ϵ was a symbol in the input alphabet. If you have good insight, you could exploit it to solve the problem more quickly, e.g. if you happen to figure out the language of the NFA, you can directly design the DFA (the language happens to be $(1^*0)^+0^*$). Otherwise, you can apply the technique for converting the NFA into a DFA, which due to the small size of the automata and small input alphabet is not too bad. The result of the transformation should look something like the solution shown in Figure 2 (under my assumption that ϵ is the empty string).

3. Lisp (10 %):

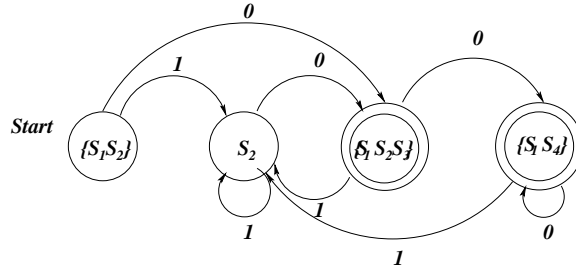


Figure 2: A Deterministic Finite Automata Solution for Problem 2c

- (a) What are the results of the following operations in Lisp (write error if it would generate an error, and say why) (5 %):

i. (CDR (MUST GET AN A))

Error, since Lisp attempts to evaluate the innermost list, and thus expects the first element in the list (MUST GET AN A) to resolve to a function.

ii. (CDR (QUOTE (MUST GET AN A)))

(GET AN A), in this case the QUOTE operator suppresses evaluation of the innermost list, and CDR gets the tail of the list (Franz Lisp calls this REST in the tutorial).

iii. (CAR (1 2 3 4))

Error, since Lisp attempts to evaluate the innermost list, and thus expects the first element in the list (1 2 3 4) to resolve to a function.

- (b) What does the QUOTE Operator do in Lisp (short answer) (5 %)?

The QUOTE operator suppresses evaluation of its arguments, which allows for example creation of lists of undefined atoms.

4. Compiler Construction Tools (30 %):

- (a) Give a brief description of what a lexical analyzer does (5 %)

The lexical analyzer discovers terminals of the language in the input stream, and communicates by returning tokens to the parser.

- (b) Give a brief description of what a parser does. (5 %)

A parser does syntactic analysis, usually it is responsible for recognizing the nonterminals and generating a parse tree representation of the input symbols (a syntax tree labeled with attributes). The parser's input is usually the scanner (lexical analyzer) output and is a stream of tokens read one token at a time.

- (c) Consider the following simple calculator language, derive pseudocode for a Flex and Bison (Lex and Yacc) style lexical analyzer and parser (you don't have to specify Bison actions) (20 %):

```

Program ::= StatementList
StatementList ::= Statement StatementList | Statement
Statement ::= Id '=' Expr | 'read' Id | 'write' Expr
Expr ::= Term | Expr AddOp Term
Term ::= Factor | Term MultOp Factor
Factor ::= '(' Expr ')' | Id | Literal
AddOp ::= '+' | '-'
MultOp ::= '*' | '/'
Id ::= Letter | Letter IDSuffix
IDSuffix ::= Letter IdSuffix | Number IdSuffix | Letter | Number

```

```

Letter ::= [A-Za-z]
Literal ::= '-' Unsigned | Unsigned | 0
Unsigned ::= Digit | [1-9] DigitString
DigitString ::= Digit DigitString | empty
Digit ::= [0-9]

```

I built a real program to do this (in only a few minutes), there were a few adjustments made to the grammar, since I used traditional BNF, I did not use a Kleene star to denote 0 or more repetitions, nor did I use the + to denote one or more repetitions, so some of the productions could be eliminated, making `Unsigned` and `Id` the terminals. The primary reason for doing this, is because we want identifiers and the magnitude part of the number to be contiguous strings of characters, trying to let the parser do that would permit white space between the symbols, as the scanner skips over this. This leads to a scanner (in flex) which looks like:

```

/* A Simple Scanner for a toy Pascal like language
   based on the example from the flex man page, but with
   some more annotations by W. A. Maniatty

```

As you can probably tell, flex uses C style comment delimiters, a flex program compiles into C traditionally (much like Unix `lex`).

The overall program structure is broken into 3 parts, separated by lines of `%%` symbols, so the program looks like:

```

<definitions>
%%
<rules>
%%
<user code>

```

```

*/

```

```

/* Define non literal symbol here */
DIGIT    [0-9]
LETTER   [a-zA-Z]

```

```

/* end of definitions section, comments are not permitted in
   the rules section which follows, so I'll put them here
   In the rules section, we describe tokens, syntax is:

```

```

<pattern_to_match> "{" <handler_code> "}"

```

The pattern to match is a regular expression using either the non-literals specified above in the definitions section or quoted character strings. For reserved words (or key words) don't use quotes.

In the handler code, the variable `char *ytext` is a pointer to a null terminated string containing the text associated with the token
`int yrlen` is the number of characters (excluding the null terminator) in the string pointed to by `ytext`

```

*/
%%

%{
#include "parser.tab.h" /* So that the parser defined tokens are
                        known by the scanner */
#include <malloc.h> /* for linux to get the declaration of free correct */
}%

{DIGIT} {
    printf( "Unsigned: <%s>=%d\n", yytext, atoi( yytext ) );
    return Unsigned;
}

[1-9]Digit+ {
    printf( "Unsigned: <%s>=%d\n", yytext, atoi( yytext ) );
    return Unsigned;
}

"read" { printf("Read <%s>\n", yytext);
         return MyRead;
}

"write" { printf("Write <%s>\n", yytext);
         return MyWrite;
}

[a-zA-Z][a-zA-z0-9]* {
    printf( "An identifier %d characters long: <%s>\n", yyleng, yytext );
    return Id;
}

"=" { printf("An Assignment operator: <%s>\n", yytext);
     return Assignment;
}

"+" { printf("PLUS <%s>\n", yytext);
     return Plus;
}

"-" { printf("MINUS <%s>\n", yytext);
     return Minus;
}

"*" { printf("MULT <%s>\n", yytext);
     return Mult;
}

"/" { printf("DIV <%s>\n", yytext);
     return Div;
}

```

```

"(" { printf("LParen <%s>\n", yytext);
      return LParen;
}

")" { printf("RParen <%s>\n", yytext);
      return RParen;
}

[ \t\n]+          /* eat up whitespace, no action */

. {
    printf( "Default action, unrecognized token/character: <%s>\n",yytext );
}

%%
/* Rules section just ended, Begin user code section */

/* All the work is done by yyparse, nothing to do in the scanner */
/* end of user code and the whole program */

and a parser which looks like:
/* Bison parser file for a simple toy pascal language */

%{
/* C declarations go here */
/* #include <FlexLexer.h> /* needed in C++ under linux to access yytext properly */
#include <malloc.h>
/* #include <stdio.h> */
/* #include <stdlib.h> */
#define YYERROR_VERBOSE /* Turn on verbose debugging output */
extern char *yytext;
%}

/* Define Tokens, that is the terminals, the following correspond to keywords
NOTE: Since some values like BEGIN have reserved meaning, I've
prefixed the names of the tokens with "MY_" to avoid conflict
*/
%token Id
%token Unsigned

%token MyRead
%token MyWrite

%token LParen
%token RParen

/* Define Operators, operators with the lowest precedence declared first
We can use literals here, but you should be careful that multicharacter
literals get double quotes (like the assignment operator), single
character literals can use single quotes (like the less than operator)
*/
%nonassoc Assignment          /* Assignment */

```

```

%left Plus Minus          /* arithmetic operators */
%left Mult Div
%right UMinus    /* Unary minus NOT USED as a token! Just for precedence */

%%

/* Nonterminals and their productions are defined here */

Program:      StatementList

StatementList: Statement StatementList | Statement

Statement :   Id Assignment Expr | MyRead Id |
             MyWrite Expr

Expr:        Term | Expr AddOp Term

Term:       Factor | Term MultOp Factor

Factor:     LParen Expr RParen | Id | Literal

/* set high precedence on unary minus */
Literal:    Unsigned | '0' |
           | Minus Literal %prec UMinus

AddOp:     Plus | Minus

MultOp:    Mult | Div

%%

/* This program is needed to handle parse errors (bad input by the user) */
int
yyerror(char *str){
    /* For verbose output, defined YYERROR_VERBOSE in the declarations section */
    fprintf(stderr, "%s: current Token <%s>\n", str, yytext);
}

int
main(int argc, char *argv[]){
    yyparse();
}

```

5. Type Systems And Memory Management (10 %)

(a) Consider the following code segment using C like syntax:

```

struct {
    double x;
    double y;
} PointA;

struct {

```

```

    double radians;
    double distance;
} PointB;

PointA.x = 1;
PointB.y = 2;
PointB radians = 1;
PointB distance = 2;

if (PointA == PointB){
    printf('PointA == PointB\n');
} else {
    printf('PointA != PointB\n');
}

```

What would the outcome be if a compiler using type name equivalence was handed this system? What about if structural equivalence was used (5 %)?

If the compiler used name equivalence checks, this program would NOT pass, as the two structs do not share a common type name. If the compiler used structural equivalence, this program would pass (although from reading the field names of the struct, perhaps it should not pass!).

- (b) Consider the following program, draw the stack and show the activation records for the program after the user types in "abc" (5 %).

```

#include <stdio.h>

void p(){
    char ch = getchar(); /* read a character of input */
    if (ch != '.')
    {
        p();
        putchar(ch); /* write a character to output */
    }
}

int
main(){
    p();
    return 0;
}

```

Assuming that the called routine is responsible for preserving its own registers, the stack and activation records should look similar to what is shown in Figure 3.

6. Miscellaneous(10 %):

- (a) Why do most computers use twos complement instead of ones complement (2 points)?
Because ones complement has two representations of the value 0 (+0 and -0), while twos complement has only one representation.
- (b) Why do compiler writers use static scope (3 %)?
Because it makes static (compile time) analysis of the code easier and allows for more efficient execution.
- (c) What does Binding Time mean (5 %)?

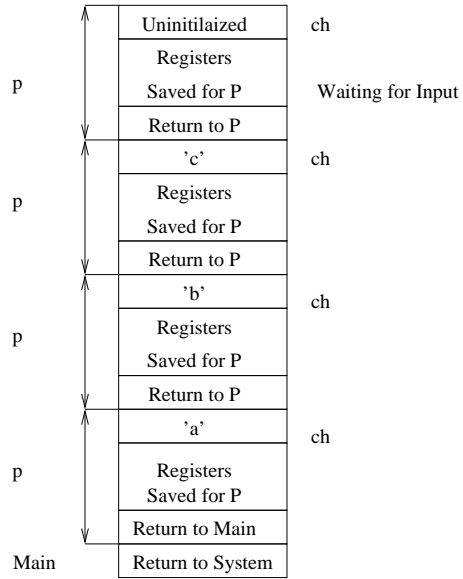


Figure 3: Activation Records and Stack Described in Problem 5b, this stack grows upward (top is up).

Binding time refers to when a programming construct receives its value. Note that some programming constructs (e.g. instruction address and constant values) are not variables, and in some cases are represented using identifiers.