

Midterm Exam
198:515 Compilers and Programming Languages I
Fall 2011
November 3, 2011
SAMPLE SOLUTION

DO NOT OPEN THE EXAM
UNTIL YOU ARE TOLD TO DO SO

Name: _____

Student ID: _____

Instructions

I have tried to provide enough information to allow you to answer each of the questions. If you need additional information, make a *reasonable* assumption, write down the assumption with your answer, and answer the question. There are **6** problems, and the exam has **8** pages. Make sure that you have all pages. The exam is worth **100** points. You have **80 minutes** to answer the questions. Good luck!

This table is for grading purposes only

| | |
|-------|-------|
| 1 | / 10 |
| 2 | / 15 |
| 3 | / 15 |
| 4 | / 10 |
| 5 | / 15 |
| 6 | / 35 |
| total | / 100 |

Problem 1 — context-free grammars (10 pts)

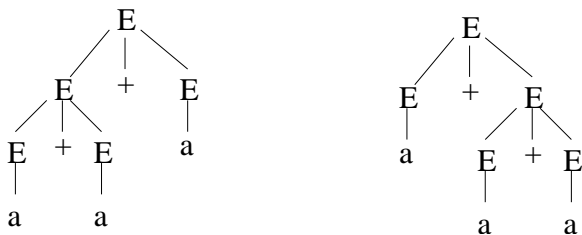
A context-free grammar (CFG) G with the start symbol E is defined as follows:

$$\begin{array}{l|l} 1 & E ::= E + E \\ 2 & E ::= a \\ 3 & E ::= b \end{array}$$

1. Show that the above grammar is ambiguous (3pts)

A grammar is ambiguous if there is a word in the language for which there are two distinct parse trees, two distinct leftmost derivations, or two distinct rightmost derivations. The following two distinct parse trees can be generated for

$a + a + a$



2. Give an unambiguous CFG G' that accepts the same language as G , i.e., $L(G') = L(G)$ (3pts).

Here is one possible grammar:

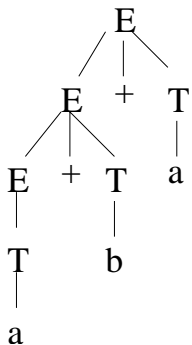
$$\begin{array}{l|l} 1 & E ::= E + T \mid T \\ 2 & T ::= a \\ 3 & T ::= b \end{array}$$

3. Show a leftmost derivation in your grammar G' for the sentence $a + b + a$ (2pts).

$E \Rightarrow_L E + T \Rightarrow_L E + T + T \Rightarrow_L T + T + T \Rightarrow_L a + T + T \Rightarrow_L a + b + T \Rightarrow_L a + b + a$

4. Show the G' parse tree for the sentence $a + b + a$ (2pts).

$a + b + a$



Problem 2 - bottom-up parsing (15 pts)

After constructing the LR(1) canonical collection, **assume** that the resulting DFA that recognizes handles includes the following states. NOTE: The states are unrelated and “artificial”.

Give the ACTION table for these four states, **ignoring the GOTO portion** in case of a shift action (just write “SHIFT” in the appropriate slot of the table). For a reduce action, give the rule itself (not its rule number). State explicitly which state has a conflict (if any), and what kind of conflict.

- $S_0 = \{ [B \rightarrow aB.b, b], [A \rightarrow a.bb, a], [A \rightarrow a.cc, b] \}$
- $S_1 = \{ [A \rightarrow a.Sab, a], [A \rightarrow a.cb, a], [A \rightarrow a.bb, b], [A \rightarrow aScb., eof], [S \rightarrow ., a] \}$
- $S_2 = \{ [B \rightarrow aB.b, eof], [B \rightarrow a.cb, a], [B \rightarrow aSc., c] \}$
- $S_3 = \{ [A \rightarrow aS., b], [B \rightarrow a.cb, eof] [B \rightarrow abb., b] \}$

LR(1) (partial) ACTION table:

| | a | b | c | eof |
|-------|--------------------------|--|--|----------------------|
| S_0 | | shift | shift | |
| S_1 | $S \rightarrow \epsilon$ | shift | shift | $A \rightarrow aScb$ |
| S_2 | | shift | $B \rightarrow aSc, \text{ shift}$ shift/reduce conflict | |
| S_3 | | $A \rightarrow aS, B \rightarrow abb$ reduce/reduce conflict | shift | |

Problem 3 - syntax-directed translation schemes (15 pts)

Assume the following expression:

```
Start ::= Expr !
Expr ::= + Expr Expr
Expr ::= * Expr Expr
Expr ::= - Expr
Expr ::= Opnd
Opnd ::= 1 | 2 | 3
```

1. Write a syntax directed translation scheme that implements an interpreter for this language. Executing your parser should write the result of the expression onto the output device using the `print` command. Use the YACC syntax to specify the syntax directed translation scheme (as in the project). (13pts)

```
Start ::= Expr !    { print ($1.val); }
```

```
Expr ::= + Expr Expr    { $$val = $2.val + $3.val; }
```

```
Expr ::= * Expr Expr    { $$val = $2.val * $3.val; }
```

```
Expr ::= - Expr    { $$val = - $2.val; }
```

```
Expr ::= Opnd    { $$val = $1.val; }
```

```
Opnd ::= 1    { $$val = 1; }
```

```
Opnd ::= 2    { $$val = 2; }
```

```
Opnd ::= 3    { $$val = 3; }
```

2. Explicitly list all attributes that you are using here, and state for each attribute whether it is inherited or synthesized (2pts).

All non-terminal symbols, except *Start*, have the synthesized attribute “val”. The attribute is integer-valued.

Problem 4 — lambda calculus (10 pts)

1. Give a λ -term that has the following properties. If you believe that no such λ -term exists, write “**does not exist**” as your answer (6pts, with 2pts each):

- The λ -term reduces to a normal form if you use a *call-by-name* reduction strategy, but does not reduce to a normal form if you use a *call-by-value* reduction strategy.

$((\lambda x. 1) ((\lambda x.x x)(\lambda x.x x)))$

- The λ -term reduces to a normal form if you use a *call-by-value* reduction strategy, but does not reduce to a normal form if you use a *call-by-name* reduction strategy.

does not exist

- The λ -term does not reduce to a normal form in neither reduction strategies.

$((\lambda x.x x)(\lambda x.x x))$

2. The **Y-combinator** $Y \equiv \lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))$ “computes” a fixed point of a function $F = \lambda f.(\dots f \dots)$. Show that $YF = F(YF)$. Hint: Use symbols F and Y to denote the corresponding lambda abstractions wherever possible. In other words, only “expand” F and Y when needed. (4pts)

$$\begin{aligned} YF &= ((\lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))) F) \\ &= (\lambda x.F(x x)) (\lambda x.F(x x)) \\ &= F((\lambda x.F(x x)) (\lambda x.F(x x))) \\ &= F(YF) \end{aligned}$$

Problem 5 — Scheme (15 pts)

1. Assume the following definition of function “test1”:

```
(define test1
  (lambda (a alist)
    (cond
      ((null? alist) '())
      ((pair? (car alist)) (cons (test1 a (car alist)) (test1 a (cdr alist))))
      ((eq? a (car alist)) (cons (cons a '()) (test1 a (cdr alist))))
      (else (cons (car alist) (test1 a (cdr alist)))))))
```

List the output of the following Scheme programs after the “→”: (3pts each)

```
(test1 'c '(c)) --> ((c))
```

```
(test1 'c '(a b c (d (e c)) e)) --> (a b (c) (d (e (c))) e)
```

```
(test1 'c '(a (b) d)) --> (a (b) d)
```

2. Assume the following definition of function “test2”:

```
(define test2
  (lambda (f x)
    (map f (map f x))))
```

List the output of the following Scheme programs after the “→”: (3pts each)

```
(test2 (lambda(x) (+ x 2)) '(3 5 8 10 1)) --> (7 9 12 14 5)
```

```
(test2 (lambda(x) (if (> x 5) (- x 1) (+ x 2))) '(3 5 8 10 1)) --> (7 6 6 8 5)
```

Problem 6 — compiler project (35 pts)

Assume that your compiler can only use the following ILOC instructions:

| Operation | | | Meaning |
|-----------|------------|-------------------|-----------------------------------|
| loadI | c | $\Rightarrow r_x$ | $c \rightarrow r_x$ |
| load | r_x | $\Rightarrow r_y$ | $\text{MEM}(r_x) \rightarrow r_y$ |
| store | r_x | $\Rightarrow r_y$ | $r_x \rightarrow \text{MEM}(r_y)$ |
| nop | | | no operation |
| add | r_x, r_y | $\Rightarrow r_z$ | $r_x + r_y \rightarrow r_z$ |
| sub | r_x, r_y | $\Rightarrow r_z$ | $r_x - r_y \rightarrow r_z$ |
| mult | r_x, r_y | $\Rightarrow r_z$ | $r_x * r_y \rightarrow r_z$ |
| output | c | | print $\text{MEM}(c)$ |

```
i: integer;
A: array[100,50] of integer;

for i= ... to ... do
begin
// begin of basic block
... = A[i-1, i+1] + 1 ; // S1
// end of basic block
end
```

You should assume

1. Addresses are byte addresses, and an integer value is stored in a 4 byte word.
2. The reserved register r_0 contains the memory offset 1024. All variables are allocated starting from this address.
3. Arrays are stored in memory in column-major order. Arrays are addressed starting with index 0, i.e., 0-based indexing is used. Example: The first column of our array A above, is accessed as $A[0,0]$ through $A[99,0]$.
4. Assume the following offset values (in terms of bytes) for the variables in our sample program:
 $\text{offset}(i) = 0, \text{offset}(A) = 4$
5. When answering the questions below, **you can only use ILOC instructions listed in the above table.**
6. Your compiler has to produce a code shape that uses a “fresh” virtual register for every computed and/or loaded value. No need to write code to check out-of-bounds array accesses.

1. Give the ILOC code sequence that a simple, one pass, basic compiler (such as the one you implemented in project 1) would generate for the statement $S1$. **Do not perform any optimizations.** (29 pts)

loadI 1024 $\Rightarrow r_0$

loadI 0 $\Rightarrow r_1$ // $r_1 = \text{offset}(i)$
 add $r_0, r_1 \Rightarrow r_2$ // $r_2 = \text{addr}(i)$
 load $r_2 \Rightarrow r_3$ // $r_3 = \text{content}(i)$
 loadI 1 $\Rightarrow r_4$ // $r_4 = 1$
 sub $r_3, r_4 \Rightarrow r_5$ // $r_5 = (i-1)$

loadI 0 $\Rightarrow r_6$ // $r_6 = \text{offset}(i)$
add $r_0, r_6 \Rightarrow r_7$ // $r_7 = \text{addr}(i)$
load $r_7 \Rightarrow r_8$ // $r_8 = \text{content}(i)$
loadI 1 $\Rightarrow r_9$ // $r_9 = 1$
 add $r_8, r_9 \Rightarrow r_{10}$ // $r_{10} = (i+1)$

loadI 100 $\Rightarrow r_{11}$ // $r_{11} = \#$ of array elements in single column
 mult $r_{11}, r_{10} \Rightarrow r_{12}$ // $r_{12} = 100 * (i+1)$ column major order
 add $r_5, r_{12} \Rightarrow r_{13}$ // $r_{13} = (i-1) + 100 * (i+1)$
 loadI 4 $\Rightarrow r_{14}$ // $r_{14} = 4$ (bytes per word)
 mult $r_{14}, r_{13} \Rightarrow r_{15}$ // $r_{15} = 4 * [(i-1) + 100 * (i+1)]$
loadI 4 $\Rightarrow r_{16}$ // $r_{16} = \text{offset}(A)$
 add $r_0, r_{16} \Rightarrow r_{17}$ // $r_{17} = \text{addr}(\text{base}(A))$
 add $r_{17}, r_{15} \Rightarrow r_{18}$ // $r_{18} = \text{addr}(A[i-1,i+1])$
 load $r_{18} \Rightarrow r_{19}$ // $r_{19} = \text{content}(A[i-1,i+1])$

loadI 1 $\Rightarrow r_{20}$ // $r_{20} = 1$
 add $r_{19}, r_{20} \Rightarrow r_{21}$ // $r_{21} = A[i-1,i+1] + 1$

2. Is there any opportunity for CSE in your code? If yes, please underline the “redundant” instructions, i.e., instructions that your CSE optimizer should eliminate. (6pts)