Indian Institute of Technology, Kharagpur

Department of Computer Science and Engineering

Mid-Semester Examination, Autumn 2016-17

Compilers (CS 31003)

Students: 112 **Date**: 14-Sep-16 (AN)

Full marks: 60 **Time**: 2 hours

Instructions: Marks for every question is shown with the question. No further clarifications to any of the questions will be provided. Make and state your assumptions if you need.

---

1. A string in Python language is a sequence of characters enclosed in quotes. Python treats single quotes the same as double quotes. However, if a string starts with a single quote (') it must end with a single quote. If it starts with a double quote (") it must end with a double quote. A string quoted by single (double) quotes may have double (single) quotes as normal characters.

   No newline is allowed within a string. However, escape characters – \s (space), \t (tab), \b (backspace), \n (new line), and \\ (backslash) – may be used. They are interpreted in the string.

   Examples of strings are:

Input / Lexeme	Lexical Value	Remarks
`'Good day'`	`Good day`	String with single quote
`"Good day"`	`Good day`	String with double quote
`'Hi, "How are you?"'`	`Hi, "How are you?"`	Double quote within single quoted string
`"I'm good"`	`I'm good`	Single quote within double quoted string
`'Good\sday'`	`Good day`	\s inserts a space
`'Good\tday'`	`Good  day`	\t inserts 4 spaces
`'Corrx\bect'`	`Correct`	\b Erases 'x'
`'Need a line here\n in next line'`	`Need a line here` `in next line`	\n inserts a newline String is placed in 2 lines
`'Do new line with \\n'`	`Do new line with \n`	Escaping \
`'String with newline` `is wrong'`	ERROR	String spans 2 lines This is an error

   Write the Flex specification to tokenize (with interpretation of escapes and quotes) strings in Python. **[15]**

   You may make suitable assumptions in the design. Clearly state your assumptions.

   Hint: *You may use Start Conditions in Flex.*

2. Consider the grammar $G =< T, N, P, S >$ where

$T =$ Set of Non-terminals $=$ { **id**, (,), +, *, **$** }

$N =$ Set of Non-terminals $=$ { E }

$P =$ Set of Production Rules $=$

1: $E \rightarrow E + E$
2: $E \rightarrow E * E$
3: $E \rightarrow (E)$
4: $E \rightarrow$ **id**

$S =$ Start Non-terminal $= E$

The LR parser table for this grammar is given by:

State	Action						GO TO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

(a) Using the table, parse the input string

id * (id + id + id) * id $

and fill up the parsing trace in the format given below. **[12]**

Step	Stack	Symbols	Input	Action
(1)	0		**id * (id + id + id) * id $**	shift
(2)	0 3	**id**	*** (id + id + id) * id $**	shift
...	...	...	...	...

(b) Using the trace, write the right-most derivation of **id * (id + id + id) * id $** according to G. **[3]**.

3. Consider the grammar

S	$\rightarrow$	**for** (E_1 ; E_2 ; E_3) S_1	E	$\rightarrow$	$E_1 = E_2$	E	$\rightarrow$	$E_1 < E_2$
S	$\rightarrow$	$A = E$;	E	$\rightarrow$	$E_1 + E_2$	E	$\rightarrow$	**num**
A	$\rightarrow$	**id** [E]	E	$\rightarrow$	E_1 ? E_2 : E_3	E	$\rightarrow$	**id**
			E	$\rightarrow$	A			

where operators and constructs follow the syntax and semantics of C. **num** is an integer constant and **id** is an identifier (integer variable). While parsing a sentence derived from this grammar, we want to translate it to an equivalent 3-address code using the following attributes and methods.

$E.type$:	Type of an expression E. This can be integer or boolean.
$E.loc$:	Location for an expression E. This is used only if $E.type$ = integer.
id.loc:	Binding location for the identifier **id**.
num.val:	Value of the numeric (integer) constant.
$E.truelist$:	List of quad's with dangling **true exits**. This is used only if $E.type$ = boolean.
$E.falselist$:	List of quad's with dangling **false exits**. This is used only if $E.type$ = boolean.
$S.nextlist$:	List of quad's with dangling **exits** for statement S.
$nextinstr$:	Global counter to the array of quad's – the index of the next quad to be generated.
$A.loc$:	Temporary used for computing the offset for the array reference by summing the terms $i_j \times w_j$, where i_j is the index and w_j is the size of the array respectively in j^{th} dimension.
$A.array$:	Pointer to the symbol-table entry for the array name. This has *base* and *type*.
	The base address of the array, say, $A.array.base$ is used to determine the actual l-value of an array reference after all the index expressions are analyzed.
$A.type$:	Type of the sub-array generated by A. For any type t, the width is given by $t.width$. For any array type t, $t.elem$ gives the element type.
$gentemp()$:	Generates a new temporary, inserts it to the Symbol Table and returns a pointer to it. Every temporary is prefixed with t and is numbered in two digits starting from 00.
$makelist(i)$:	Creates a new list containing only i, an index into the array of quad's.
$merge(p_1, p_2)$:	Concatenates the lists pointed to by p_1 and p_2 and returns a pointer to the resultant list.
$backpatch(p, i)$:	Inserts i as the target label for each of the quad's on the list pointed to by p.
$emit(result, arg1, op, arg2)$:	Spits: `result = arg1 op arg2`. `op` usually is a binary operator. If `arg2` is missing, `op` is unary (prefix or postfix). If `op` also is missing, this is a copy instruction.

(a) Augment the grammar with ϵ-productions and rewrite the production rules as needed for designing the semantic actions. State the attributes for the new non-terminals that you introduce. Also list if you need more methods. **[5]**

(b) Write the semantic actions for the production rules above. For every rule show how the attributes are updated using the methods given. Clearly state any assumption that you make. **[10]**

Note:

- *An* **id** *is of type* integer *and a* **num** *is an integer constant. Hence an expression E, by default, has an $E.type$* = integer. *This may change to* boolean *when E is used in the condition in* **for** *construct or in* **?:** *operator.*

(c) Using your semantic actions, and the following context

```
int a[] = { 1, 0, 2, 0, 0, 3 }, i;
```

translate

```
for (i = 0; i < 6; i = i + 1)
    a[i] = a[i] ? 0 : i;
```

into 3 address codes. Assume that the context (declarations) have already been processed and the Symbol Table is in the following state when the translation of the `for` construct starts:

Name	Type	Size	Offset
a	array(6, integer)	24	0
i	integer	4	24

[2+3+8+2 = 15]

i. Draw the parse tree.

ii. Annotate the symbols on the parse tree with the attributes to show the flow of your semantic computation.

iii. Using the annotated parse tree and the *emit* method, generate the quad's starting from 100.

iv. Show the updates (entries of temporary variables) on the Symbol Table.