CS3025 Compiladores

Examen 1
Pregrado
2024-1

Nombre del docente: Igor Siveroni

Alumno:

Apellidos: _____ Nombres: _____

Sección: _____ Fecha: _____ Nota:

Indicaciones:
La Duración es de 90 minutos.
La evaluación consta de 5 preguntas.

Pregunta 1 (4 puntos) Expresiones Regulares y autómatas

a) (1 pt) Definimos a los números enteros como cadenas de 1 o más dígitos, sin ceros a la izquierda. Escribir las expresiones regulares que definen a:

- Los números enteros.

- A los números enteros pares.

b) (1.5 pts) Escribir una expresión regular que describa al lenguaje de fechas escritas en el formato `dd/mm/yyyy`, donde `dd`, `mm` y `yyyy` corresponden a el número de día, mes y año, respectivamente. La longitud de `dd` y `mm` debe de ser 2, para lo cual se deberán usar 0's a la izquierda si es necesario p.ej. 06 en lugar de 6. Los años tendrán longitud máxima de 4 y NO podrán tener 0's a la izquierda.

c) (0.5 pt) Escribir una expresión regular que describa a los literales de tipo *string*, es decir, a las cadenas que empiezan y acaban con comillas (") p.ej "Hola". Asumir que dentro de la cadena puede ir cualquier carácter con la excepción de las comillas.

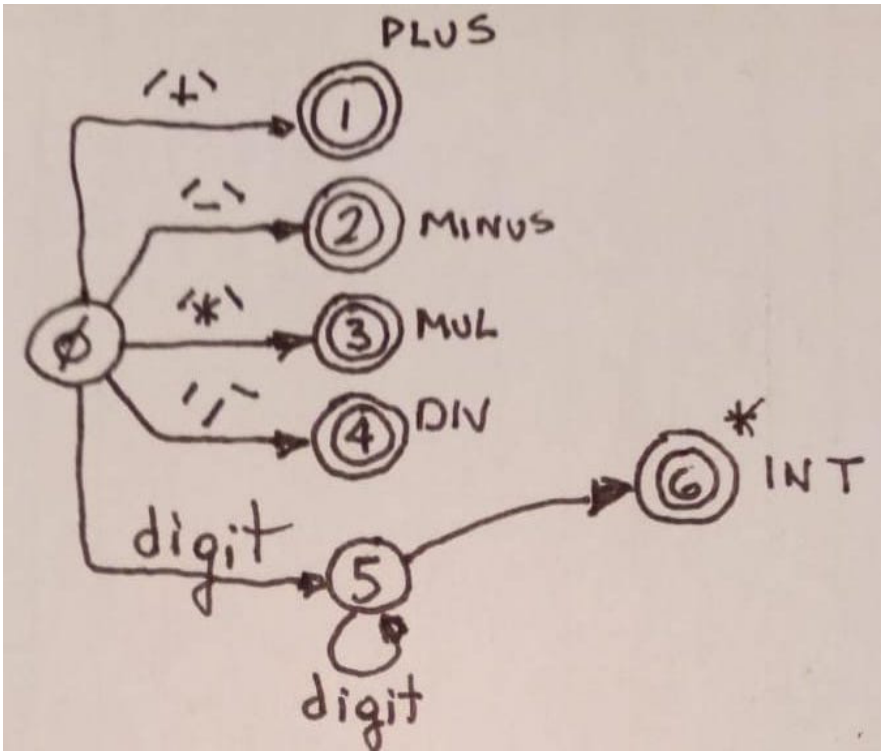
d) (1 pt) Escribir un autómata finito determinista (AFD) que acepte identificadores que empiecen con un carácter alfanumérico seguido de cero o más caracteres alfanuméricos, numéricos o el carácter '_', con la condición de que el carácter '_' no puede aparecer al final de la cadena ni más de una vez de forma consecutiva. Por ejemplo, `x`, `x2_4` y `x_y_8` son identificadores válidos, pero `x__24`, `x22_` y `x23__` no lo son.

Pregunta 2 (4 puntos) Análisis Léxico

El autómata y código presentados en esta pregunta describen e implementan a un analizador léxico para un lenguaje que necesita reconocer tokens para los operadores "+", "-", "*" y "/" (tokens PLUS, MINUS, MUL, DIV), y números enteros (INT).
Se nos pide extender el lenguaje para que acepte un nuevo operador, "++", representado por el token PPLUS, y al conjunto de números de punto flotante, denotado por el token FLOAT.

a) (0.5 puntos) Los números de punto flotante FLOAT validos en nuestro lenguaje están compuestos de una parte entera, definida por las **mismas** reglas que INT, seguida de un punto decimal y 0 o más dígitos.
Escribir las expresiones regulares que definen a INT y FLOAT.

b) (2 puntos) Extender el AFD de abajo para que reconozca PPLUS (++) y FLOAT. No es necesario re-dibujar el AFD, solo se pide indicar de manera clara las modificaciones. Reutilizar los estados ya existentes en la medida de lo posible, teniendo en consideración que algunos estados podrían dejar de ser estados finales. Numerar a los nuevos estados a partir del número 7.



c) (1.5 puntos) El método `void nextToken()` de abajo pertenece al analizador léxico (scanner) especificado por el AFD **inicial** de la parte (b) . Extender el scanner con los estados y transiciones necesarios (código) para reconocer PPLUS y FLOAT, de acuerdo con los cambios introducidos en la parte (b). Solamente agregar las modificaciones al código inicial y los puntos en los que hay que agregar nuevo código.

```
Token* Scanner::nextToken() {
  Token* token;
  char c;
  state = 0;
  startLexema();
  while (1) {
    switch (state) {
    case 0: c = nextChar();
      if (c == ' ') { state = 0; startLexema(); }
      else if (c == '\0') return new Token(Token::END);
      else if (c == '+') state = 1;
      else if (c == '-') state = 2;
      else if (c == '*') state = 3;
      else if (c == '/') state = 4;
      else if (isdigit(c)) { state = 5; }
      else return new Token(Token::ERR, c);
      break;
  case 1: return new Token(Token::PLUS);
  case 2: return new Token(Token::MINUS);
  case 3: return new Token(Token::MUL);
  case 4: return new Token(Token::DIV);
  case 5: c = nextChar();
    if (isdigit(c)) state = 5; else state = 6; break;
  case 6: rollBack();
    return new Token(Token::INT, getLexema());
    }
  }
}
```

Pregunta 3 (4 puntos). Gramáticas libres de Contexto

Consideremos la siguiente gramática que describe al lenguaje de expresiones aritméticas.

```
Exp → Exp + Exp        Exp → Exp - Exp        Exp → num
Exp → Exp * Exp        Exp → Exp / Exp
```

donde `num` denota los números enteros.

a) (1.75 puntos) Escribir la derivación por la izquierda, y el árbol sintáctico correspondiente, que produce la palabra `3*2+10/2`. Indicar los valores del token `num`. Además, indicar el resultado (número) de evaluar la expresión usando el árbol sintáctico producido.

b) (1.75 puntos) Escribir la derivación por la derecha, y el árbol sintáctico correspondiente, que produce la palabra `3*2+10/2`. Indicar los valores del token `num`. Además, indicar el resultado (número) de evaluar la expresión usando el árbol sintáctico producido.

c) (0.5 punto) ¿La gramática es ambigua? ¿Por qué?

Pregunta 4: Gramáticas (4.5 puntos)

a) (3.25 pts) La gramática LP define a un subconjunto de las fórmulas de la lógica proposicional. LP está compuesta de las siguientes reglas:

```
LP → true | false    LP → LP (and | or | implies) LP
LP → not LP          LP → '(' LP ')'        LP → id
```

donde `not`, `implies`, `and` y `or` son operadores booleanos.

LP es ambigua. Escribir una versión NO ambigua de LP de tal manera lo siguiente se cumpla:

- El operador unario `not` tiene mayor precedencia que todos los demás operadores.
- El operador `implies` tiene mayor precedencia que `and` y `or` y, además, asocia por la derecha.
- Los operadores `and` y `or` tienen igual precedencia entre sí y asocian por la izquierda.

Escribir la gramática sin usar el operador * (cerradura de Kleene / Kleene closure).
La nueva gramática puede tener recursividad por la izquierda (puede ser left-recursive).

b) (1.25 puntos) Palíndromes
Un palíndrome o capicúa es una cadena que se lee igual de izquierda a derecha que de derecha a izquierda (s = reverse(s)).

- Escribir la gramática de los palíndromes de 0's y 1's que tienen como centro al carácter especial $. Estas cadenas tendrán siempre longitud impar y mayor o igual que 1.

```
Palindrome ::=
```

- Implementar la función `parsePalindrome()` del analizador léxico de descenso recursivo que reconoce los palíndromes definidos por la gramática anterior.

Pregunta 5: Descenso Recursivo y Acciones Semánticas (3.5 puntos)

La gramática de un subconjunto de programas de la máquina virtual de pila SVM es la siguiente:

```
P :: = LInstr (SEMICOLON LInstr)*        LInstr ::= [LABEL] Instr
Instr ::= PUSH NUM | POP | DUP | ADD | SUB
```

donde las palabras en mayúsculas indican nombres de Tokens (con las interpretaciones obvias). Escribir un parser de descenso recursivo (solo las funciones indicadas abajo) que reconozca programas P y que, a través del uso de acciones semánticas, modifique los contenidos de la pila de acuerdo con las operaciones especificadas en `Instr`. Asumimos la existencia de una pila cuyos contenidos solo podrán ser leídos y modificados mediante la siguiente API:

- int stack_pop(): Remueve y retorna el último elemento de la pila.
- int stack_top(): Retorna el último elemento de la pila (sin modificar la pila).
- int stack_height(): Retorna la altura de la pila.
- void stack_push(int n): Coloca n encima de la pila.

Asumimos que el programa empieza con una pila vacía. Las acciones semánticas deberán verificar que la altura de la pila es la correcta antes de realizar las operaciones. Si no es así, imprimir un mensaje y abortar. De igual modo, el parser abortara (`exit`) si el input no satisface la gramática. Usar el API de los parsers desarrollados en los laboratorios.

```
// Retornar el último elemento de la pila
int parseP() {




}

void parseInstr {
```

```
void parse LInstr {


   return;
}
```

Código ejemplo del uso del API usado para implementar Parsers de descenso recursivo incluimos el código de abajo.

```
int Parser::parseExpression() {
  int accum, v;
  accum = parseTerm();
  while(match(Token::MINUS) || match(Token::PLUS)) {
    Token::Type op = previous->type;
    v = parseTerm();
    if (op == Token::PLUS) accum += v;
    else accum -= v;
  }
  return accum;
}

int Parser::parseTerm() {
  int accum, v;
  accum = parseUnary();
  while(match(Token::MULT) || match(Token::DIV)) {
    Token::Type op = previous->type;
    v = parseUnary();
    if (op == Token::MULT) accum *= v;
    else accum /= v;
  }
  return accum;
}

Int Parser::parseUnary() {
  int sign = 1;
  if (match(Token(Minus))
    sign = -1;
  return sign*parseFactor();
}

int Parser::parseFactor() {
  if (match(Token::NUM)) {
    return stoi(previous->lexema);
  }
  if (match(Token::LPAREN)) {
    int v = parseExpression();
    if (!match(Token::RPAREN)) {
      cout << "Expecting right parenthesis" << endl;
      exit(0);
    }
    return v;
  }
  cout << "Couldn't find match for token: " << current << endl;
  exit(0);
}
```