Contents

En el curso vamos a aprender a procesar de manera automatica información, es decir hacer un compilador.

```
    | 1er parcial 30: 85 examen, 15 tareas
70% | 2do parcial 30: 85 examen, 15 tareas
    | 3er parcial 40: 85 examen, 15 tareas

30% | Proyecto final
```

Ni de chiste sube mas de 30 por mucho que le pongas ganas

Trabajando bien el primer y segundo parcial ya podriamos decir que aprobamos el curso. No descuidar los parciales.

Un conjunto es una enumaeración de elementos, hay conjuntos y subconjuntos. Un conjunto numerable es aquel en el que podemos establecer relaciones 1 a 1 con el conjunto, el dominio de los numeros reales. Ej. los meses del año.

Aunque un conjunto sea infinito puede ser numerable porque los reales son infinitos. El conjunto de numeros reales no es enumerable porque entre cada numero hay infinidad.

Se muestra un conjutno de la forma:

$$D = \{x \mid x \text{ is decimal digit}\}$$

En la figura de arriba tenemos una forma de definir conjuntos de forma no enumerada.

Hay operaciones entre los conjuntos que requieren de mayores especificaciones, por ejemplo:

$$D = \{x \mid (x = a) \vee (x = b) \vee (x = 1)\}$$

Donde $\vee$ significa `or`. Asi que tenemos implicados multiples operadores como $\vee$.

Se permite que un conjunto este formado por otros conjuntos. Se dice que B es un subconjunto de A cuando todos los elementos de B estan en A.

1. Indexar un conjunto

Si tenemos $A = \left\{A_{s_1}, A_{s_2}, ...\right\}$

- Intersección: Si elemento esta en A y B
- Union: Si el elemento esta en A o B

Se puede hacer intersección de intersecciones de la forma $\cup_{i=1}^{n} A_i$ si tenemos $A_1, A_2, A_3, ...A_n$

La diferencia devuelev los elementos de A que no estan en B, los elementos que le faltan a B para ser identico a A

El producto cartesiano son todas las tuplaes que podemos formar con los elementos definidos en los multiples elementos

Tarea: ver qué cosa es Ro, hacer los ejercicios

2. Relaciones

Las funciones son, por ejemplo, un caso particular de las relaciones, donde se rekacione elementos de un dominio con elementos de una imagen.

Hay relaciones que no se pueden expresar con relaciones aritméticas, como es el caso de las funciones.

Hay relaciones que no se pueden expresar con relaciones aritméticas, como es el caso de las funciones.

Si x y y están relacionados por una relación R, ambos pertenecen a R se expresa como xRy

- El **dominio** de una relación son las x tales que existe una $x, (x, y) \in S$
- El **rango** de una relación S son las y tales que existe una $x, (x, y) \in S$

Relaciones entre relaciones

Si x y y están relcionados con la intersección de RyS

Si tachamos una relación S se dice que *no estan en la realción*, $\neg S$

Reflexivo es que a la realción no le hace nada ...

Una relación es simétrica si para cada (x, y), x está relacionado con y y lo mismo viceversa, lo que habitualmente le llamamos función compuesta.

Seguir viendo

Hay relaciones que no se pueden expresar como una función, como el caso de la relación que se expresa como una tabla. Otro ejemplo son las relaciones del algebra booleana, donde se expresa utilizando tablas de verdad.

Hay relaciones que se expresan con gráficas, las flechas que podemos ver indican el sentido.

$xRz \land zRx \land y \land Z$

Supongamos que tenemos una relación R que relacion de X a Y y una ralción de S de Y a Z. Una relación compuesta de $R \cdot S$ es igual a la union de ambas relaciones, donde desaparece el intermedio Y, ver el ejemplo de grafos del libro, figura 2-2.

Asociatividad: La asociatividad en las relaciones puede hacer que quitemos los paréntesis.

Relaciones de orden superior

$R^2 = R \cdot R$ es paracticamente R compuesta con R

Para hacer de siguientes ordenes ocupamos los de los anteriore.

3. Relación R^+

Es la unión de los elementos en la.

Ver definicion 2-1

, la relación de cerradura. Si tengo un automata, estamos en un punto de todos los elementos que se puede alcanzar en un paso. R2 es todos los que podemos alcanzar por dos pasos.

Ver el algoritmo de Warshall

, se usa para encontrar la cerradura de conjuntos A. n es cantidad de filas, en P obtenemos el resultado.

$R\neg\neg S$ si R y S no estan realcionados, ... ?

Exercises 2-1.2

4. String

Definimos las cadenas de caracteres como un conjunto $V = \{a, b, c, ...\}$ llamado vocabulario, un ejemplo de V son los caracteres del alfabeto inglés o lo mismo pero griego.

La concatenación de dos caracteres es solo un par “ab”. La compuesta es “aba”

Una cadena son todas las secuencias de todos los tamaños que se pueden hacer con los elementos de los vocabularios.

Entonces si $V = \{a, b, c\}$ entonce $V^2 = \{\text{aa}, \text{ab}, \text{ac}, ...\}$

V^* es la cadena vacía unión de V^+. V es la cedana vacia, un conjunto que no tiene caracteres.

Una cadena de caracteres, si tienes V, la cadena de caracteres es $V^* = V_{\text{vacio}} \cup V^+$

5.

Una cosa es tener coche y manejarlo a ser experto de F1. Lo mismo pasa en el mundo tech.

Chomsky llega a explicar los tipos de lenguajes:

1. Gramatica regular, no hay ninguno
2. Gramatica independiente del contexto.
3. Gramatica dependiente del contexto. Ejemplo «lo contó en el banco», depende del contexto donde se usa para tener su significado. Aqui estan los lenguajes de programación.
4. Gramaticas sin restricciones, no hay reglas que puedan describir el lenguaje. Ejemplo las Islas Canarias. En la isla de la gomera usaban chiflidos para enviar mensajes, el lenguaje silvado de la gomera. El lenguaje silvado o silvo de la gomera.

Las definiciones de esto lo vemos la proxima semana

6. Teoría de Conjuntos

Un lenguaje de programación consiste de una conjunto de programas (o *strings*). Como veremos, es conveniente usar una gramatica como un vehiculo formal para generar estos programas. Diversas relaciones pueden ser definidas con las reglas de una gramatica, y estas relaciones pueden llevar a la compilación de forma efectiva de los algoritmos del lenguaje asociado con la gramatica. El objetivo de esta sección es de introducir los elementos básicos de los conjuntos y las cadenas (*strings*). En particular, los conceptos de conjunto estan introducidos en la Sección 6.1. La noción de una relación se describe en la Sección 7. Otros puntos importantes sobre las relaciones incluidas en esta sub-sección son las de grafos y *transitive closure* de una relación. La sección termina con un tratamiento elementario de los *strings*.

6.1. Conceptos básicos

Cuando decimos *conjunto* nos referimos a una collección de objetos de cualquier tipo. La palabra *objeto* se usa con un sentido muy amplio, donde se incluye también objetos abstractos. Un concepto fundamental de la teoria de conjuntos es la de la *pertenencia* a un conjunto. Cualquier objeto que pertenezca a un conjunto es llamado un *miembro* o *elemento* de dicho conjunto.

Si un elemento p pertenece a un conjunto A, se escribe de la forma $p \in A$, lo cual se lee como «p es un elemento del conjunto A» o «p pertenece al conjunto A». Si contamos con un objeto q que no pertenece a un conjunto A, entonces expresamos este hecho como $q \notin A$.

6.1.1. Definición de conjuntos

Existen muchas maneras de especificar un conjunto. Por ejemplo, un conjunto que consiste de los digitos decimales se escribe, por lo general, de la forma $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, donde los nombres de los elementos se rodean en llaves y se separan por comas. Si deseamos marcar el conjunto como D podemos hacerlo de la forma $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

6.1.1.1. Predicados

Sin embargo, nombrar cada elemento no siempre es conveniente, por eso podemos optar por describir, por ejemplo a D, de la forma: $D = \{x \mid x \text{ es un dígito decimal}\}$. En este caso el símbolo $\mid$ se lee «tal que», lo que sigue al símbolo es el predicado, en nuestro ejemplo «es un dígito decimal».

Si decimos que $P(x)$ es cualquier predicado, entonces $\{x \mid P(x)\}$ define un conjunto y se lee de la forma «el conjunto de todas las x tal que $P(x)$ es verdadero». Un elemento a pertenece a un conjunto si $P(a)$ es verdadero; de lo contratio a no pertenece al conjunto. Si denotamos el conjunto $\{x \mid P(x)\}$ como B, entonces $B = \{x \mid P(x)\}$.

Los conjuntos que son especificados listando sus elementos también se pueden describir por medio de un predicado. Por ejemplo, el conjunto $\{a, b, 1, 9\}$ se puede definir como $\{x \mid (x = a) \vee (x = 1) \vee (x = 9)\}$. El símbolo $\vee$ denota una disyunción lógica o `or`.

6.1.1.2. Conectores y operadores

Un predicado puede contener los conectores lógicos $\wedge$ (conjunción lógica o `and`), $\vee$ (disyunción lógica o `or`), $\neg$ (negación lógica), $\rightarrow$ (si entonces), $\Leftrightarrow$ (si y solo si), y los operadores relacionales $<$, $\leq$, $=$, $\neq$, $\geq$, y $>$. Además, un predicado también puede contener el cuantificador existencial ($\exists$, que significa «existe») y el cuantificador universal ($\forall$, representando «para todos»).

6.1.1.3. Series

Aunque es posible describir cualquier conjunto con un predicado, también se suele especificar con métodos, por ejemplo $C = \{1, 3, 5, ...\}$ y $S = \{a, a^2, a^3, ...\}$. En estos ejemplos se puede determinar los elementos restantes en base a los elementos presentes y su contexto.

6.1.2. Conjuntos finitos e infinitos

El número de elementos distintos presentes en cualquier conjunto puede ser finito o infinito. Llamamos a un conjunto *finito* si es que contiene un número finito de elementos distinguibles; en cualquier otro caso se dice que es *infinito*.

6.1.3. Subconjuntos

Antes de continuar, hay que destacar que en la tería de conjuntos no hay restricciones sobre los objetos que pueden ser miembros de un conjunto. No es raro tener conjuntos cuyos miembros son en sí conjuntos, tal como pasa con $A = \{0, \{a, b\}, 1, \{p\}\}$. Eso si, hay que saber distinguir entre el conjunto $\{p\}$, el cual es un elemento de A, y el elemento p, el cual es un miembro de $\{p\}$, mas no es miembro de A.

Para dos conjuntos cualesquiera A y B, si cada elemento de A es un elemento de B, entonces A es llamado un *subconjunto* de B, o también se dice que A está *incluido* en B, o bien, que B incluye A. Simbólicamente, esta relación se denota con $A \subseteq B$ (que se lee «A es un subconjunto de B, o es igual a B»), o de forma equivalente como $B \supseteq A$ (que se lee «B es un superconjunto de A, o es igual a A»).

6.1.4. Igualdad

Cabe aclara que para dos conjuntos cualquiera A y B, que $A \subseteq B$ no implica que $B \subseteq A$, excepto en el caso de que ambos sean iguales, lo que se escribe de la forma $A = B$ y que requiere que $A \subseteq B$ y $B \subseteq A$.

6.1.5. Subconjunto propio

Se le dice, a un conjunto cualquiera A, *subconjunto propio* de B si $A \subseteq B$ pero $A \neq B$. A esta relación se le representa de la forma $A \subset B$.

6.1.6. E y $\emptyset$

Ahora introducimos dos conjuntos especiales; el primero incluye todos los conjuntos sobre los que se discute (E), mientras que el segundo está incluido en todos los conjuntos sobre los que se discute ($\emptyset$). Se dice *conjunto universal* (E) a aquel conjunto que incluye todos los conjuntos involucrados en la discusión. Por ejemplo, el conjunto universal para los numeros naturales puede ser $E = \{0, 1, 2, 3, ...\}$. Un conjunto que no contiene ningun elemento se llama *conjunto vacío* o *conjunto nulo*, de denota se la forma $\emptyset$.

Dado cualquier conjunto A, sabemos que el conjunto nulo $\emptyset$ y el conjunto A son ambos subconjuntos de A, es decir $A \subseteq A$ y $\emptyset \subseteq A$. También sabemos que para cualquier elemento $p \in A$, el conjunto $\{p\}$ es un subconjunto de A, $\{p\} \subset A$. De forma similar, podemos considerar otros subconjuntos de A. Algo mejor que estar manejando subconjuntos individuales de A, podemos buscar describir hechos para todos subconjuntos de A.

Es una verdad *verdad vacuba* que $\emptyset \subseteq A$ para cualquier conjunto A debido a que cada elemento de $\emptyset$ ($\{\}$) es también un elemento de A. O acaso podemos nombrar un solo miembro de $\emptyset$ que no sea mimebro de A, en ese caso la condición se rompería.

6.1.7. Conjunto potencia

Para cada conjunto A, a la collección o familia de todos los subconjuntos de A se le conoce como *conjunto potencia* de A. El conjunto potencia o *power set* de A se denota como $p(A)$ o 2^A, es decir $p(A) = 2^A = \{x \mid x \subseteq A\}$, nótese como $\subseteq$ implica que A mismo está en 2^A.

6.1.8. Conjunto indexado y conjunto índice

Ahora introducimos el concepto de un *conjunto indexado*. Si $J = \{s_1, s_2, s_3, ...\}$ y A es la familia de conjuntos $A = \left\{A_{s_1}, A_{s_2}, A_{s_3}, ...\right\}$ tal que para cada $s_i \in J$ le corresponde un conjunto $A_{s_i} \in A$, y también $A_{s_i} = A_{s_j}$ si y solo si $s_i = s_j$. Es entonces A llamado un *conjunto indexado*, J es un *conjunto indice*, y cualquier subíndice, tal como s_i en A_{s_i}, es llamado un *índice*.

Una familia indexada de conjuntos también se puede escribir de la forma $A = \{A_i\}_{i \in J}$. De forma particula, si $J = \{1, 2, 3, ...\}$, entonces $A = \{A_1, A_2, A_3, ...\}$. También, si $J = \{1, 2, 3, ..., n\}$, entonces tendremos $A = \{A_1, A_2, A_3, ..., A_n\} = \{A_i\}_{i \in I_n}$ donde $I_n = \{1, 2, 3, ..., n\}$. Para un conjunto S que contiene n elementos, el conjunto potencia $p(S)$ se escribe como el conjunto indexado $p(S) = \{B_i\}_{i \in J}$ donde $J = \{0, 1, 2, ..., 2^n - 1\}$.

$$p(S_n) = \left\{S_{n_i}\right\}_{i \in J} \text{ donde } J = \{0, 1, 2, ..., 2^n - 1\}$$

6.2. Operaciones

Consideremos ahora operaciones que se pueden realizar con conjuntos. Nos enfocaremos en las operaciones de intersección, unión y complemento. Uno puede emplear estas operaciones para construir nuevos conjuntos al combinar los elementos de los conjuntos dados.

> **Nota**: Aunque en el libro se le dice colecciones a conjuntos de conjuntos, para hacer más amena la lectura les llamaré *familias* a todos los conjuntos de conjuntos, a menos que especifique lo contrario; Por ejemplo, en el caso en que lea explícitamente *familia de subconjuntos* de un conjunto A (2^A). Esto lo hago porque entiendo las palabras conjunto y collección como forma para referirse a lo mismo, a conjuntos.

6.2.1. Intersección

La intersección de dos conjuntos cualquiera A y B, escrito de la forma $A \cap B$, es el conjunto que consiste de todos los elementos que pertenecen tanto a A como a B, es decir

$$A \cap B = \{x \mid x \in A \land x \in B\}$$

Podemos tener operaciones de intersección para cualquier familia indexada de conjuntos $A = \{A_i\}_{i \in J}$. De esta forma podemos obtener todos los elementos que son miembros de todos los conjuntos de la familia de conjuntos:

$$\bigcap_{i \in J} A_i = \{x \mid x \in A_i \text{ para toda } i \in J\}$$

Si se trara de una familia indexada de conjuntos de tamaño n, entonces podemos escribirlo de la forma:

$$\bigcap_{i=1}^{n} A_i = \bigcap_{i \in I_n} A_i = A_1 \cap A_2 \cap ... \cap A_n \text{ donde } I_n = \{1, 2, ..., n\}$$

6.2.1.1. Conjuntos desarticulados

Se dice que dos conjuntos A y B están desarticulados (*disjoint*) si, y solo si $A \cap B = \emptyset$, es decir, si no hay elementos en común. A una familia se le conoce como *familia desarticulada* si, para cada par posible de conjuntos en la familia, los dos conjuntos están desarticulados, es decir, que no hay elementos en común entre ninguno de los conjuntos de la familia. A los elementos de una familia desarticulada, es decir, a los conjuntos de la familia, se dice que están *mutuamente desarticulados*.

Si A es una familia indexada $A = \{A_i\}_{i \in J}$. La familia A es una *familia desarticulada* si, y solo si $A_i \cap A_j = \emptyset$ para todas las $i, j \in J$ done $i \neq j$. Que es lo mismo que describimos antes, no hay ningún elemento en común entre los conjuntos miembros de la familia.

6.2.2. Unión

Para dos conjuntos cualquiera A y B, se le dice *unión*, escrito de la forma $A \cup B$, al conjunto de todos los elementos que están presentes en cualquiera de los conjuntos A y B.

$$A \cup B = \{x \mid x \in A \lor x \in B\}$$

Para cualquier familia indexada $A = \{A_i\}_{i \in J}$ podemos hacer lo mismo que con la intersección, obtener el conjunto de la unión de todos los conjuntos miembros de la familia, un conjunto que contiene todos los elementos presentes en por lo menos uno de todos los miembros de la familia:

$$\bigcup_{i \in J} A_i = \{x \mid x \in A_i \text{ en al menos uno con } i \in J\}$$

En el caso de una familia indexada con un tamaño n podemos escribir

$$\bigcup_{i=1}^{n} A_i = \bigcup_{i \in I_n} = A_1 \cup A_2 \cup ... \cup A_n \text{ donde } I_n = \{1, 2, ..., n\}$$

6.2.3. $A - B$ Complemento relativo (diferencia)

El complemento relativo de B en A (o lo que es lo mismo, de B respecto a A), se puede escribir de la forma $A - B$, que es el conjunto que contiene todos los elementos de A que no son miembros de B. Al complemento relativo de B en A también se conoce como *diferencia* de A y B.

$$A - B = \{x \mid x \in A \land x \notin B\}$$

6.2.4. $E - A$ Complemento absoluto

El complemento relativo de un conjunto A en E (el conjunto universo), es decir $E - A$, es llamado el *complemento absoluto* de A

6.3. 2-Tupla o sequencia

Hasta ahora nos hemos enfocado en conjuntos, su igualdad, y las operaciones que podemos realizar para producir nuevos conjuntos. Ahora introduciremos la noción de una pareja ordenada

Una pareja ordenada, sequencia o 2-tupla consiste de dos objetos en un orden establecido. No es lo mismo hablar de una pareja ordenada y de un conjunto de dos elementos. El orden de los dos objetos es importante. Se escribe una 2-tupla de la forma (x, y).

La igualdad de un par de 2-tuplas (x, y) y (u, v) se define con $(x, y) = (u, v) \Leftrightarrow ((x = u) \land (y = v))$, es decir, que la igualdad de ambas 2-tupla es equivalente lógicamente a que x es igual a u y y a v.

6.3.1. Implica

Se dice que A implica B, escrito de la forma $A \Rightarrow B$, si y solo si $A \to B$ siempre es verdadero. En matematicas se usan $A \to B$ y $A \Rightarrow B$ como iguales.

6.4. n-tupla

El concepto de un par ordenado puede extenderse para definir trios ordenados, y en general cualquier una *n-tupla* de n elementos ordenados. Escribimos una n-tupla como $(x_1, x_2, ..., x_n)$.

6.5. $A \times B$ Producto cartesiano

Un concepto importante en la teoría de conjuntos es la del producto cartesiano. Dados dos conjuntos A y B, se le conoce como *producto cartesiano* de A y B al conjunto de todas las 2-tuplas en las que el primer miembro de cada tupla es un elemento de A y el segundo miembro es un elemento de B. Se escribe de $A \times B$.

$$A \times B = \{(x, y) \mid x \in A \land y \in B\}$$

Asi como el resto de operaciones en conjuntos, podemos extender el del producto cartesiano a familias de conjuntos. Si $A = \{A_i\}_{i \in I_n}$ es una familia indexada de conjuntos donde el conjunto indice $I_n = \{1, 2, ..., n\}$. Entonces podemos escribir el producto cartesiano de los conjuntos de la familia como:

$$\bigtimes_{i \in I_n} A_i = A_1 \times A_2 \times ... \times A_n$$

Podemos definir el producto cartesiando de forma resursiva como:

$$\bigtimes_{i \in I_1} A_i = A_1,$$

$$\bigtimes_{i \in I_m} A_i = \left(\bigtimes_{i \in I - (m-1)} A_i \right) \times A_m \text{ para m} = 2, 3, ..., \text{n}$$

Basados en lo descrito, si el siguiente bloque de codigo devuelve un iterador sobre las 2-tuplas generadas entre dos conjuntos:

```rust
pub fn times<'a, U, V>(a: &'a [U], b: &'a [V]) -> impl Iterator<Item = (U, V)> + 'a
where
    U: Clone,
    V: Clone,
{
    a.iter()
        .map(|a| b.iter().map(|b| (a.clone(), b.clone())))
        .flatten()
}
```

Entonces, el bloque de código equivalente a la definición de producto cartesiano «recursivo» queda:

```rust
pub fn fam_times<T: Clone>(a: &[&[T]]) -> Vec<Vec<T>> {
    let mut res = Vec::new();

    for i in 1..a.len() {
        res = match i {
            1 => times(a[0], a[1]).map(|e| vec![e.0, e.1]).collect(),
```

```
            2.. => times(&res, a[i])
                .map(|(mut vec, val)| {
                    vec.push(val);
                    vec
                })
                .collect(),
            _ => unreachable!("Range is 0..n"),
        };
    }

    res
}
```

Esta definición de producto cartesiano de n conjuntos se relaciona con la definición de n-tuplas en el sentido de que: $A_1 \times A_2 \times ... \times A_n = \{(x_1, x_2, ..., x_n) \mid (x_1 \in A_1) \land (x_2 \in A_2) \land ... \land (x_n \in A_n)\}$

6.6. Ejercicios

1. Da otra descripción a los siguientes conjuntos e indica aquellos que son infinitos.

   $a) \{x \mid x \text{ es entero y } 5 \leq x \leq 12\} = \{5, 6, 7, 8, 9, 10, 11, 12\}$

   $b) \{2, 4, 8, ...\} = \{x \mid x \bmod 2 = 0\} = \{2x \mid x \in \mathbb{N}\}$, es infinito

   $c)$ Todos los paises del mundo $= \{\text{México}, \text{Estados Unidos}, \text{Canada}, ...\}$

2. Dado que $S = \{2, a, \{3\}, 4\}$ y &R = {{a}, 3, 4, 1}&, indica si los siguientes puntos son verdaderos o falsos.

$a)$ Falso	$\{a\} \in S$
$b)$ Verdad	$\{a\} \in R$
$c)$ Verdad	$\{a, 4, \{3\}\} \subseteq S$
$d)$ Falso	$\{\{a\}, 1, 3, 4\} \subset R$
$e)$ Falso	$R = S$
$f)$ Verdad	$\{a\} \subseteq S$
$g)$ Falso	$\{a\} \subseteq R$
$h)$ Verdadero	$\emptyset \subset R$
$i)$ Verdadero	$\emptyset \subseteq \{\{a\}\} \subseteq R \subseteq E$
$j)$ Falso	$\{\emptyset\} \subseteq S$
$k)$ Falso	$\emptyset \in R$
$l)$ Verdad	$\emptyset \subseteq \{\{3\}, 4\}$

3. Muestra que $(R \subseteq S) \land (S \subset Q) \rightarrow R \subset Q$ siempre es verdadero. Es correcto reemplazar $R \subset Q$ por $R \subseteq Q$?

   Si $R \subseteq S$ entonces tenemos dos escenarios posibles, en donde $R \subsetneq S$ y donde $R = S$. El *peor* de los casos es cuando $R = S$, pues es donde más elementos puede tener S, facilitando la violación de la condición.

   Si $R = S$, entonces sabemos que si $S \subset Q$, por lo tanto $R \subset Q$ pues se tratan del mismo conjunto en esencia.

   Es correcto remplazar $R \subset Q$ por $R \subseteq Q$ pues ambos implican que R es subconjunto de Q, aunque uno deja la posibilidad de que ambos sean iguales. Sin embargo, si se sabe que $R - Q \neq \emptyset \Rightarrow R \neq Q$, entonces es conveniente escribirlo de la forma $R \subset Q$.

4. Obten $p(S)$ donde S es:

   $a)\ p(\{a, \{b\}\}) = \{\emptyset, \{a\}, \{\{b\}\}, \{a, \{b\}\}\}$

   $b)\ p(\{1, \emptyset\}) = \{\emptyset, \{1\}, \{\emptyset\}, \{1, \emptyset\}\}$

   $c)\ p(\{X, Y, Z\}) = \{\emptyset, \{X\}, \{Y\}, \{Z\}, \{X, Y\}, \{X, Z\}, \{Y, Z\}, \{X, Y, Z\}\}$

5. Dado que $A = \{x \mid \text{es entero y } 1 \leq x \leq 5\}$,
6. Muestra las identidades:
    1. $A \cap A = A$

$$\text{Si } A = \{x \mid P(x)\}, \text{ donde} P(x) \text{es un predicado que evalúa a 1 si es verdadero}$$
$$\begin{aligned} A \cap A &= \{x \mid P(x) \wedge P(x)\} \\ &= \{x \mid P(x)[1 \wedge 1]\} \\ &= \{x \mid 1P(x)\} \\ &= \{x \mid P(x)\} = A \end{aligned}$$

    2. $A \cap \emptyset = \emptyset$

$$A = \{x \mid P(x)\}, \emptyset = \{x \mid x \notin E\}$$
$$\begin{aligned} A \cap \emptyset &= \{x \mid P(x) \wedge x \notin E\} \\ &\text{Si } E = A \\ A \cap \emptyset &= \{x \mid P(x) \wedge x \notin \{x \mid P(x)\}\} \\ &= \{x \mid P(x) \wedge \neg P(x)\} \\ &= \{x \mid P(x)(1 \wedge \neg 1)\} \\ &= \{x \mid 0P(x)\} \\ &= \{x \mid 0\} \end{aligned}$$

Y como el predicado es siempre 0, el conjunto estará vacío

7. Relaciones

El concepto de una relaciòn es un concepto básico en matemáticas, asi como en el dia a dia. Asociado El concepto de una relaciòn es un concepto básico en matemáticas, asi como en el dia a dia. Asociamos a *relación* está el acto de comparar objetos que estan relacionados de una forma u otra. La habilidad de una computadora para realizar distintas tareas basada en el resultado de una comparación es otro punto importante empleado durante la ejecución de un programa típico. En esta sub-sección formalizaremos el concepto de relación y discutiremos los métodos empleados para representar una relación usando una matriz o su grafo. La matriz de relación es muy útil para determinar las propiedades de una relación y también para representar una relación en una computadora. Algunas propiedades básicas de una relEntre a de laación stTambién se mueasi como ciertas clases imporantes introducidas.

La palabra *relación* sugiere algunos ejemplos familiares de relaciones tal como la relación padre a hijo, hermana a hermano, tio a sobrino. Los ejemplos familares en la aritmética son las relaciones tales como *menor que*, *mayor que*, y demás relaciones de igualdad entre dos números. También conocemos entre la relación entre en area de un trángulo equilatero y el tamaño de uno de sus lados o el area de un cuadrado y el tamaño de uno de sus lados. Estos son solo unos de los ejemplos de relaciones entre dos objetos.

á el acto de compa rar objetos que estan relacionados de una forma u otra. La habilidad de una computadora para realizar distintas tareas basada en el resultado de una comparación es otro punto importante empleado durante la ejecución de un programa típico. En esta sub-sección formalizaremos el concepto de relación y discutiremos los métodos empleados para representar una relación usando una matriz o su grafo. La matriz de relación es muy útil para determinar las propiedades de una relación y también para representar una relación en una computadora. También se muestran algunas de las propiedades básicas de una relación asi como ciertas clases imporantes introducidas.

La palabra *relación* sugiere algunos ejemplos familiares de relaciones tal como la relación padre a hijo, hermana a hermano, tio a sobrino. Los ejemplos familares en la aritmética son las relaciones tales como *menor que*, *mayor que*, y demás relaciones de igualdad entre dos números. También conocemos entre la relación entre en area de un trángulo equilatero y el tamaño de uno de sus lados o el area de un cuadrado y el tamaño de uno de sus lados. Estos son solo unos de los ejemplos de relaciones entre dos objetos.

A lo largo de la discusión consideraremos las relaciones, llamadas *relaciones binarias*, entre pares de objetos. Cualquier conjunto de pares ordenados define una relación binaria. Llamamos a una relación simplemente relación. A veces es conveniente expresar un objeto particular ordenado, digamos $(x, y) \in R$, donde R es una relación, de la forma xRy.

En matematicas las relaciones son denotadas de forma típica por símbolos especiales en lugar de letras mayúsculas (ejemplo A). Un ejemplo familiar es la relación «menir que» para los números relaes. Una relación esta denotado por $<$. De hecho, $<$ debería ser considerado como el nombre de un conjunto donde sus elementos son 2-tuplas. Más precisamente, la realción $<$ es $<= \{(x, y)|x, y \in \mathbb{R} \wedge x < y\}$

7.1. Dominio y Rango

Digamos que S es una relación binaria. El conjunto $D(S)$ contiene todos los objetos x tales que para alguna y, entonces $(x, y) \in S$ se le llama *domino*, es decir:

$$D(S) = \{x \mid (\exists y)((x, y) \in S)\}$$

En donde el símbolo $\exists$ denota qualidad de existencia. De forma similar $R(S)$ es el conjunto de todos los objetos y tales que para algun elemento x, se cumple $(x, y)nS$, a este conjunto se le llama el *rango* de S, es decir:

$$R(S) = \{y \mid (\exists x)((x, y) \in S)\}$$

7.2. Operaciones

Debido a que se ha definido a una relación como un conjunto de 2-tuplas, es por lo tanto aplicar las operaciones usales de los conjuntos a los conjuntos de relaciones por igual. El conjunto resultante también estará compuesto por 2-tuplas que define algunas relaciones. Si R y S son dos relaciones, entonces $R \cap S$ define una relación tal que $x(R \cap S)y \Leftrightarrow xRy \wedge xSy$

8. Gramáticas

8.1. Apuntes

Una gramatica esta definida por una 4-tupla (V_n, V_t, S, φ). Tiene un conjunto V_n y V_t donde ambos son distintos a $\emptyset$. V_n se conoce como conjunto de no terminales, en cambio V_t se conoce como conjunto de los terminales.

Definimos a $V = V_t \cup V_n$, también conocido como el simbolo distinguido o inicial de la gramática, donde $S \in V_n$

φ es un conjunto de reglas de la gramatica, que tiene de la formula:

$(V_t \cup V_n)^* V_n (V_t \cup V_n)^* \to (V_t \cup V_n)$

Recorcemos que $*$ es el conjunto de todas las posibilidades.

Puso en ejemplo, un identificador es una secuencia de caracteres que puede tener tales caracteres, no tiene espacios y puede empezar con a-Z o _, se usa para todo tipo de entes como variables, estructuras, etc.

Chomsky define, por poner un ejemplo, al identificador como

$$V_n = \{I, LD\}(\text{No terminales})$$
$$V_t = \{a, b, c, ..., z, 0, 1, 2, ..., 9\}(\text{Terminales})$$
$$S = I(\text{Simbolo distinguido})$$
$$\varphi = \{I \to L, I \to IL, I \to a|b|c|...|z, D \to 0, D \to 1, ..., D \to 9\}(\text{Reglas de producción})$$

Donde $I \to L$ quiere decir que I deriva en L. Entonces I es el indificador de la gramatica, podemos cambiar esa I por elementos hasta que en la parte derecha solo aparezcan identificadores $I \to L \to a$, o $I \to IL \to LL$, notemos como en el último paso remplazamos I por L, pues en las reglas definidas en φ especificamos que podemos sustituirlo, siempre y cuando no se trate de un digito, ejemplo $I \to ID \to IDD \to LDD$.

Es pues la gramatica anterior lo que define formal identificador. Si queremos darle soporte, por ejemplo a otro caracter como _ entonces simplemente lo agregamos a V_t.

8.2. Derivación directa

Aqui no termina, describe lo que se llama *derivación directa*, si tenemos una gramática $G = (V_N, V_T, S, \Phi)$, cuando se dice que $\sigma, \varphi \in V$, donde σ es una cadena de caracteres que tiene los caracteres permitidos. Decimos que σ deriva de «phi» si ?, por lo tanto $\varphi_1, \varphi_2 \in V *$

$y = \phi_1 \alpha \phi_2$

$r = \phi_1 \beta \phi_2$

$\alpha \to \beta e \Phi$

Si $I \to IL$, entonces $\varphi_1 I \varphi_2 \to \varphi_1 I \varphi_2 L$

Una derivación directa es hacer un cambio con una regla de la gramática.

Ver los ejemplos del libro

8.3. Producción

A aplicar multiples veces una derivación directas se le conoce como porducción.

La cadena `si` produce `r`. $\Sigma(\Rightarrow) + r.\sigma \Rightarrow () + r$, se pone `+` porque implica que se hizo por lo menos una derivación, es decir pasamos por una infinidad de sigmas (y) hasta poder llegar a simga (r) que es lo mismo a $y_1 \Rightarrow y_2 \Rightarrow ...r$

Si no derivamos nada entonces queda que si = r.

Un conjunto de producción que parten siempre del simbolo distinguidor que va haciendo producciones hasta que no queden $\sigma \in V_n$. Entonces un lenguaje $L(G) = \{\sigma \mid S^* \Rightarrow \sigma \land \sigma \in V_t^*\}$, entonces es el conjunto de cadenas? tales que todas parten del signo identificador y terminan en elementos que no pertenecen a los terminales.

Ahora nos mostró una foto de Von Neumann y Openheimer que desarrollaron la primera computadora, con fines militares. Ahora nos muestra a la abuelita Cobol, Grace Hopper, que

8.4. Lenguajes

Todas las gramaticas son subconjuntos de otra.

8.4.1. Sin restricciones

Hay gramaticas que son sin reglas, no hay manera de escribir un sistema de reglas para ellas. Casi todos los lenguajes que hablamos son del tipo 1, **gramaticas sin restricciones**. Si no podemos describir con reglas en lenguaje no podemos escribir un compilador para el mismo.

8.4.2. Dependientes del contexto

Antes estan las gramaticas tipo 2, **dependientes del contexto**, por ejemplo el español, recordemos el ejemplo de «lo contó en el banco». Estas gramaticas tienen reglas de producción $\alpha \to \beta$ cumplen que la cantidad de elementos en alpha tiene que ser menor/igual que en beta ($|\alpha| \leq |\beta|$) y además $\alpha_1 \beta \in V^+$, es decir que puede tener en la parte derecha tiene que tener terminales. Tenemos que encontrar primero el terminal para ver que regla aplicar.

Porque es una gramatica dependiente del contexto, no se refiere a la fórmula, a la definición, aino quw quiere que interpretemos la definición para ver qué pasa.

La gramatica que estamos viendo del documento genera cadenas del tipo $a^n b^n c^n \mid n \geq 1$, es decir, genera cadenas de n caracteres *a*, n cantidad de *b_s y n canidad de* c_s.

Pone enfasis en que V^+ viene acompañado de $\cup$, lo que implica que se puede cumplirse o no. Por ejemplo una regla $abc \to BBB$.

Las gramaticas dependientes del contexto causan que cambiar de lugar los *statements* se logran efectos distintos.

8.4.3. Independientes del contexto

El unico cambio es que $\alpha \in V_N$, y $\beta \in V^*$. Es decir, que en la cadena α no deben existir no terminales. Por ejemplo la gramatica $\{a^n ba^n |\ n \geq 1\}$.

8.4.4. Gramaticas regulares

Lo ideal es, para Chomsky, que las gramaticas sean de tipo 4, gramaticas *regulaes*

Donde α menor que β y eso pero con una condición, $\alpha === V_N$, es decir, que existe UN SOLO terminal. β a la vez debe ser de la forma

En una gramatica regular, segun entiendo, no deben coincidir el numero de terminales, pero si debe concidir con lo que nos dice.

8.4.5. Ejemplo

En el ejemplo los no terminales se escriben con parentesis cuadrado, los or separan entre reglas. La regla de factor tiene un terminal, parentesis un no terminal y al final un parentesis, lo que lo hace no ser una gramatica regular, no cumple las reglas.

Ahora toca ver si es independiente del contexto, es decir ver que alhpa no tenga terminales, el modulo y que beta tenga cualquiera. Se cumple, por lo tanto es independiente del contexto.

Buscando *ejemplos de gramaticas.*

8.5. Definición de expresión regular (def 4.1, pag 169):

Una expresión regular es aquella donde las expresiones pueden ser contruidas usando las reglas:

1. ϕ es una expresión regular que denota un conjunto vacío
2. ε es una expresión regular que denota que el lenguaje consiste solamente de cadenas vacías, es decir $\{\varepsilon\}$
3. a, donde $a \in V_T$ es una expresión regular que denota un lenguaje consistiendo de un simbolo a es decir el lenguaje $\{a\}$
4. If e_1 y e_2 son expresiones regulares denotando los lenguajes L_1 y L_2, entonces:
   - $(e_1)|(e_2)$ es una expresión regular que denota $L_1 \cup L_2$
   - $(e_1)(e_2)$ es una expresión regular de denota $L_1 L_2$
   - $\{e_1\}$ es una expresión regular que denota L_1^*, es decir que viene e_1 que se repite 0 a n veces

La lógica en las expresiones es la misma que en los conjuntos, primero van paréntesis, luego concatenación $(*)$ y por último `or` $(|)$, similar a $\cup$ y $\vee$ pues.

Los *tokens* de un lenguaje de programación pueden ser definidos en términos de expresiones regulares y lenguajes regulares.

De ahora en adelante, cuando sea mínimo uno y después repetido le llamaré $\{a\}^+$ y es igual a $\{a\}a$

8.6. Definición de Aceptores/Autómatas de estado finito determinístico

También conocidos como autómatas deterministas finitos, un aceptor, para cualquier estado y entrada de caracteres tiene máximo una transición de estado, si no hay transición de estado definido, entonces la cadena, la entrada se rechaza como inválidad.

Es una 5-tupla (K, V_T, M, S, Z) donde:
- K es un conjunto finito no vacio de elementos llamados estados.
- V_T es un alfabeto llamado el alfabeto de entrada
- M es un mapeo del conjunto $K \times V_T$ a K, o sea que mapea (a, b) donde $a \in K, b \in V_T$ con elementos de K, es decir, dado un estado y un elemento de entrada en V_T, pasamos a un estado en K.
- $S \in K$, se llama un estado inicial.
- $Z \subseteq K$ es un conjunto no vacio llamado estados finales.

Son finitos porque el conjunto de estados es finito, son deterministicos porque cada estado mapea a otro.

Definición 4-n Lo unico que cambia en este otro autómata es que no va de K a K, sino que va de K a cualquier subconjunto de K, es decir puede ir a multiples estados.

Si uno tiene muchas entradas, es decir una cadena $S = Tt$, entonces T es una cadena y t es un caracter. EN este caso se mueve del estado

8.7. Auómata finito no determinista (pag 174)

Un autónoma finito no determinista es dificil de representar porque requiere que exploremos todas las definiciones para poder probar algo.

La unica diferencia es que M mapea $K \times V_T$ con $x \in 2^K$, es decir, que un valor de entrada $x \in V_T$ en un estado $k \in K$ no solamente deriva en un estado de K, sino que mapea a múltiples estados en K tal que puede llevar a 2^K

$$A^a \xrightarrow{\quad a \quad} B^b \xrightarrow{\quad b \quad} C$$

Figura 1: Del capítulo *Scanners*, página 176

En este tipo de autómatas, la única manera de probar que una cadena será aceptada por este tipo de autómatas es probando todas las rutas posibles, no podemos terminar directamente pues puede que no hayamos cubierto todas las rutas posibles.

8.7.1. Movimiento

Cuando decimos $M(A, ab) = \{A, B\}$, queremos dar a entender que desde el estado A, vamos a obtener todos los estados a los que podemos llegar con una cadena `"ab"`, específicamente usando el caracter `a`, en el caso de la figura Figura 1, podemos llegar tanto al estado A, como al B, lo que recordemos que hace en sí al autómata no determinado, pues mapea a más de un estado en cualquiera de sus relaciones en M.

En el caso de la máquina de Figura 1 queda como $M(A, ab) = \{A, B\} \rightarrow M(\{A, B\}, b) = \{B, C\}$, en un momento de buscar recursivamente llegamos a C, que es un estado final, y por lo tanto la cadena `"ab"` es aceptada.

Podemos describir este movimiento con:

- Si no entra nada (ε) se queda en el estado Q
- Si entra algo entra a TODOS, a la union de todos los estados P a los que podemos llegar con el inicio de la entrada $\bigcup_{P \in M(Q,T)}$, devolviendo un $M(P, t)$, en este enunciado, P no es más que el conjunto de estados donde podemos movernos desde el inicio de la cadena.

Es facil identificar un autómata no determinista viendo si devuelve un conjunto el mapeo en M o no.

Tratemos de programar el autómata en la figura 4-7, pagina 177

8.8. Autómatas finitos con transiciones vacías ε, es decir no deterministas (pag 180, def 4-4)

Sin nosotros hacer nada el autómata cambia de estado, esto mismo es lo que los hace no deterministas. Se dice que es no determinista pues si no recibe nada se puede quedar en su estado original A o puede irse a otro(s) estados $K - \{A\}$ En el vocabulario se debe incluir ε

8.9. Cerradura vacia de un estado ()

En un automata no determinista, es el conjunto de todos los estados que se pueden lograr con puras transiciones vacias.

En la tabla 4-5, ese autómata puede ir por transiciones vacias a q_1 y el estado inicial q_0, es decir $\{q_0, q_1\}$

8.10. Equivalencia entre gramatica regular y un autómata (pag 183. def 4-5)

El chiste es convertir las gramáticas en autómatas, porque eso lo podemos ejecutar en la computadora.

Hacer el ejercicio de escanear cada token de un lenguaje, sin nada de hacer espacios, aunque vengan sin ellos.

Una gramática se puede convertir en autómatas finitos no deterministas con transiciones vacías, ese autómata a su vez se puede convertir en automatas finitos deterministas.

Para lograr la transformación se pueden usar las formulas de Thompson,

El primer ejercicio del examen es, dada una gramática, debemos clasificarla. Dado una gramática que ya es regular obtener la expresión regular asociada. Una expresión regular, debemos de obtener el automata finito no determinista con transiciones vacias, y de ahi el automata finito determinista, y con esto obtener una gramática regular

De cualquier autómata finito determinista hay asociada una gramática regular que podemos obtener a partir de él, eso viene definifo en el mismo cuerpo de 4-5.

En la figura 4-11 estan las 3 formlas de Thompson. Tengo un ejemplo en el cuarderno, pongamos todos los vacios necesarios, no afectan, el problema viene cuando faltan vacíos.

Las formulas nos dejaron con un automata finito no determinista con transiciones vacias.

Hay un ejemplo en mi cuaderno, el autómata que hicimos no esta simplificado, tiene estados que sobran, no los reduzcamos, la proxima clase es ver como obtener una grámatica regular a partir del autómata finito determinista.

8.11. Ejercicio

Ejercicio del capitulo 4, tipo examen pag 172, 5

Para sacar una gramática regular del automata, para cada transicion escribimos una regla

En la conversion del Automata FInito no determinista al autómata finito determinista no garantiza que sea reducido.

Tareas:
- Equipo_N

Examenes:
- Nombre_Apellido

8.12. Flex

Le da tristeza que en las universidades no le dan atención a esto cuando hay muchas tareas que puede realizar Flex perfectamente.

Flex es un *scanner*, reconoce texto y permite extraer *tokens*.

Pregunta de examen: De donde sacamos el manual de flex? De GNU>doc>flex>version>flex.html

Un archivo de flex es un archivo de texto plano que tiene una estructura:
- Definiciones
- Reglas
- Código de usuario

Separando entre las secciones con el símbolo %%

Todos los comandos y expresiones de flez se deben de escribir en la columna 1

Un archivo de flex es un archivo de texto plano que tiene una estructura:
- Definiciones
- Reglas
- Código de usuario

Separando entre las secciones con el símbolo %%

Todos los comandos y expresiones de flez se deben de escribir en la columna 1, es importante tener control sobre los caracteres que se usan porque pueden influir en si flex correrá bien o no.

La carpeta donde están los ejecutables está en GNUWin32>bin, ahi esta flex, bison, sugar, etc. Esa carpeta tiene que estar puesta en las variables del entorno, en el `$PATH`.

Cuando agregamos una variable tenemos que guardarlo y reiniciar el equipo.

La estructura del archivo se ve de la forma:

```
definiciones
%%
reglas
%%
codigo
```

Para nuestros ejercicios solo vamos a usar 3 operaciones regulares, no vamos a usar las opciones más poderosass, flex puede tener infinitas expresiones regulares, que hace que busque en el texto de entrada todas las expresiones y decide verificar cuál hace match, las reglas:

- La que más caracteres empareje de la entrada es la que se dispara
- La que primero salga entre todas las que tienen el mismo número de caracteres

Malic, programación en C++

Ejemplo de un programa:

```
/*
     Sección de definiciones

     Todo el codigo que queremos al inicio del programa va al inicio entre
     corchetes y llave
*/
%{
     #include <stdio.h>
     unsigned int num_lines = 0, num_chars = 0;
%}

/*
     Reglas, cuenta todas las lineas y caracteres
     PATRON(expr-reg)         ACCION(codigo c++)
*/
%%
\n        {
               ++num_lines; // Encontramos una nueva linea
               ++num_chars; // Encontramos un caracter
          }
.         ++num_chars; // Matchea todo menos nueva linea
%%

int main() {
     yylex(); // Función predefinida que invoca Flex

     // Podemos acceder a las variables estáticas que declaramos
     printf( "# of lines = %d, # of chars = %d\n", num_lines, num_chars);
}
```

Flex como es Open Source y Windows tiene sus cosas hay que evitar caracteres raros. La extensión normal de los archivos de flex es `.lex`.

8.13. Invocar flex

Podemos ver las opciones de flex con `flex -h`, hay que tener cuidado porque hay ciertos caracteres que no son vàlidos para flex, como caracteres que se ven iguales a un paréntesis pero en realidad no lo son.

El codigo generado, `lex.yy.c`, contiene todo lo que escribimos, podemos ver de entre todo lo generado nuestro código:

```
case 1:
/* rule 1 can match eol */
```

```
YY_RULE_SETUP
#line 16 "flex.lex"
{
   ++num_lines; // Encontramos una nueva linea
   ++num_chars; // Encontramos un caracter
}
YY_BREAK
```

Podemos ver el `#line N`, que son instrucciones del pre-compilador de flex qué cosa agregar en este lugar de otro archivo. Lo primero que haremos es ver cómo hacer que flex no use instrucciones de pre-compilador.

Para lograrlo debemos usar `-L`, que hace que se ponga directamente el código, copiado. Ahora se muestra:

```
case 1:
/* rule 1 can match eol */
YY_RULE_SETUP
{
   ++num_lines; // Encontramos una nueva linea
   ++num_chars; // Encontramos un caracter
}
YY_BREAK
```

8.14. Compilar VSCode

Creamos un nuevo programa de C++ para consola, “Proyecto Vacio de Consola”, todos los proyectos los pondrá ahi de ahora en adelante, en la ruta personalizada que él indico.

En archivos de origen que agregue un elemeno existente, ahi tenemos que agregar el `lex.yy.lex`, hacerlo de esta forma hace que sea un símbolo que apunta al archivo dado. Después de agregarlo simplemente compila.

8.14.1. Warnings

Le salió un error que dice que la función `fileno` está *deprecated*, no conviene ponerse a pelear con este tipo de errores., para evitar esto deshabilita los warnings que molesten en la sección de definiciones, por ejemplo:

```
#pragma warning(disable:4996 6011 6385 4013)
```

Hace lo mismo con la 6011, 6385, etc. De forma que solo se muestre el error en VSCode Studio. Luego de hacer esto tiene que volver a recompilar con flex y por último recompilar en VSCode

8.14.2. Error

Hay un error que `yywrap` no está definida, entonces tenemos que hacerlo:

- Definir una función `yywrap` con la nomenclatura que pide flex
- Deshabilitarlo con `%option noyywrap`

> El proyecto hay que entregarlo sin warnings ni errores, se deshabilitan los warnings siempre y cuando sean en el código de flex, si es de nuestro código entonces debemos arreglarlo.

Con las modificaciones de supresión de errores y warnings es:

```
/*
     Sección de definiciones
     Todo el codigo que queremos al inicio del programa va al inicio entre
     corchetes y llave
*/
%{
     #include <stdio.h>
     unsigned int num_lines = 0, num_chars = 0;
     #pragma warning(disable:4996 6011 6385 4013)
%}

/* Quitar funcion yywrap */
%option noyywrap

/*
```

```
     Reglas, cuenta todas las lineas y caracteres
     PATRON(expr-reg)         ACCION(codigo c++)
*/
%%
\n        {
               ++num_lines; // Encontramos una nueva linea
               ++num_chars; // Encontramos un caracter
          }
.         ++num_chars; // Matchea todo menos nueva linea
%%

int main() {
     yylex(); // Función predefinida que invoca Flex

     // Podemos acceder a las variables estáticas que declaramos
     printf( "# of lines = %d, # of chars = %d\n", num_lines, num_chars);
}
```

O, en lugar de des-habilitar `yywrap` hay que agregar la función directamente:

```
/*
     Sección de definiciones
     Todo el codigo que queremos al inicio del programa va al inicio entre
     corchetes y llave
*/
%{
     #include <stdio.h>
     unsigned int num_lines = 0, num_chars = 0;
     #pragma warning(disable:4996 6011 6385 4013)
%}

/* Quitar funcion yywrap */
%option noyywrap

/*
     Reglas, cuenta todas las lineas y caracteres
     PATRON(expr-reg)         ACCION(codigo c++)
*/
%%
\n        {
               ++num_lines; // Encontramos una nueva linea
               ++num_chars; // Encontramos un caracter
          }
.         ++num_chars; // Matchea todo menos nueva linea
%%

/* Agregamos aqui la firma esa */

int main() {
     yylex(); // Función predefinida que invoca Flex

     // Podemos acceder a las variables estáticas que declaramos
     printf( "# of lines = %d, # of chars = %d\n", num_lines, num_chars);
}
```

Probemos a hacer este programa nosotros jaa. El termina de insertar información con `CTRL-Z` para windows.

Hubo un error en el ejercicio que hicimos entre todos, debio quedar:

$$\{a\}(b\{b\}c|a) \equiv \{a\}b\{b\}c \mid a\{a\}$$

8.15. Flex 2

Supongamos que queremos hacer un ejemplo sencillo, dado un texto, donde hay números enteros, queremos reconocer los números enteros y sumarlos.

`yywrap`, cuando encuentra el final del archivo invoca yywrap para poder realizar acciones al terminar todo el procesamiento de la entrada.

Por default, flex lee los caracteres del flujo de entrada, el `stdin`. Podemos hacer que el ejecutable tome como flujo de entrada el archivo, podemos hacerlo desde el OS con `<<`.

O podemos hacer lo siguiente, notemos que se salta el primer argumento.

```
int main(int argc, char* argv[]) {
    ++argv; // Se salta uno
    --argc;
    if (argc > 0) {
        yyin = fopen(argv[0], "r");
    } else {

    }
    yylex();
}
```

Lo que vamos a hacer es que vamos a copiar ambos a donde deben estar en VS Code. Tiene Proyetco/ aqui lo puso

En visual studio > proyecto > tiene nivel advetencia 3

El hace que en los dos lugares esté igual

En VSCode, para pasar argumentos en el debugger debemos: Proyecto > Propiedades > Depuración > Argumentos de Comandos

Tip para subir el examen:
1. Quitamos toda la basura
   - Borrar `.vs`
   - Borrar `Debug`
   - Dejar solo la carpeta del Proyecto > Projecto y el `.sln`
2. El archivo pdf va a la altura de `.sln`
3. Nombre `FelixMartinez`

Si hay errores de versiones Proyecto > Propiedades > Escogemos la pasada

Ejercicio, tengo 5 dolares, el profe quiere que hagamos uno donde se pueda escribir tengo 5 dolares, 3 pesos, debo un dolares, debe de poder hacer la suma en dólares y pesos.

8.16. Sintaxis, Scanning y Semántica

Terminamos con el scanner, que puede sacar los componentes (*scanning*), pero ahora cómo verificar que están en orden, que sirven una gramática (*sintaxis*), que verifica que las cadenas sigan las reglas del lenguaje.

Aun asi puede estar bien en componentes y sintaxis pero que no sea *semánticamente* correcto.

La alternativa es usar fuerza bruta, que hace que todas las posibilidades verifiquen si es correcto o no. El algoritmo prueba caracter por caracter todas las reglas para ver si hacen *match*.

Gramaticas recursivas por:
- $S \to Sab$: Recursividad por la izquierda, puede convertirse a recursividad por la izquierda.
- $S \to abS$: Recursividad por la derecha

8.17. Dudas

1. De la tarea ejercicio 1 y 2, si es solo numeros enteros o también flotantes.

   A:

2. Para errores cosas como numeros mal escritos, identificadores mal escritos

3. De la 3 y 4 consideramos palabra como solo letras y números, o cualquier cadena de caracteres que no contenga espacios, saltos de linea ni tabs

   A: El 3 que si tenga numeros y eso y en el 4 que solo sean palabras de caracteres

4. Los pipes de las reglas,

C -> a C|a solo se puede sacar el pipe en ese caso {a}a {a}|emptyset, como {a} incluye emptyset, podemos dejar solo repeticiones de {a}

Ejemplo, este otro tiene la solucion directa: B -> B b|c c{b}, no es una gramatica regular porque la parte derecha debe comenzar con un terminal y despues un no terminal

Le dio un error, asi que quitar el 4996 y el otro. Marca esas funciones como errores porque ella fijo eso

ctrl Z para final en el de Windows

Para pasar archivos externos:
- F10 para debug

8.18. Fuerza Bruta

Es un algoritmo que para cosas pequeñas está bien. Requiere gramaticas independientes del contexto, además de que está el problema en que se le presenten gramáticas recursivas por la izquierda.

Lo que hace la fuerza bruta es:
1. Sustituye la primera regla

$$E \to E + T|T$$
$$\Rightarrow E + T$$

Y ahi si sigue dividiendo inifitamente. El caso de la grmática de la pagina «For example, the familiar set of rules for» es recursiva por la izquierda. Pag 214

Podemos generar nuevas gramáticas no recursivas por la izquierda

Pag 214. El primer ejercicio del segundo parcial es sobre convertir de recursivo por la izquiero al reverso.

Entonces tiene

$$E \to E + T|T$$
$$T \to T \times F|F$$

Donde β_1 es T y α_1 es T y en donde β_1 es F y α_1 es F. Y pone una nueva regla...

$E \to T \mid TE'$

Y crea $E' \to +T \mid +TE'$

Para la segunda, elimina –T -> T times F|F–:

$$T \to F \mid FT'$$
$$T' \to *F \mid *FT'$$

Y como resultado tenemos una gramática que no es dependiente del contexto y que genera lo mismo mientras que es recursivo por la derecha.

Tarea:

Pag 215. Partiendo de la gramática que genera expresiones con suma, multiplicacion y paréntesis:

$$E \to T$$
$$E \to TE'$$
$$E' \to +T$$
$$E' \to +TE'$$
$$T \to F$$
$$T \to FT'$$
$$T' \to *F$$
$$T' \to *ET'$$
$$F \to (E)$$
$$F \to a$$

El algoritmo de fuerza bruta construye un arreglo `NT` de tipo `String` con los no terminales, por ejemplo `[E, E', T, T'. F]`.

Tambien tiene un arreglo de estructuras llamado `LHS` de la forma `(max, first)`, donde
- `max` es el numero de producciones por cada no terminal de NT.
- `first` indica dónde está la sustitución, en el arreglo de cadenas de los valores de sustituciòn en RHS.

Con `RHS: T, TE', +T, +TE', F, FT', *F, *ET, (E), a`

```
NT  MAX  FIRST
E   2    1 (pos 1 en el arreglo de RHS)
E'  2    3
T   2    5
T'  2    7
F   2    9
```

Adicional a estos 3 arreglos, necesita una pila llamada `hist` que va a almacenar la regla que de cada no terminal se está usando en ese momento. Usando `nohist` tenemos el valor en la posición y tenemos `sent` donde se ponen todas las opciones que se han ido probando, aso como `symb` que es lo que se quiere probar.

El algoritmo del libro tiene un error en el caso 7

Para verificar si toca hacer una sustitución revisa primero si se trata de un no terminal iterando sobre `NT`, si se encuentra en la lista entonces se va a la primera regla del arreglo de RHS

Hay otro arreglo de string que tiene los elementos de la derecha, las producciones.

En el libro hay un error porque E y T tienen dos reglas.

El algoritmo asume que:
- Los no terminales y los terminales son de solo un caracter
- Después en la tarea toca hacer que pueda ser lo que sea y unirlo con Flex

Para hacer el algoritmo de fuerza bruta tenemos una demostraciòn más formal en la 212. Vamos a representar cada estado del autòmata con ña tupla (S, I, α, β), donde:
- S: Son los estados por los que pasa el autómata. Toma valores en el conjunto de q, b, q
  - q expande reglas para adelante
  - b esta retrocediendo, en backtrack
  - t esta en un estado terminal
- I es un apuntador a la cadena de entrada por reconocer, es aqui que se asume que todos los caracteres son solo 1 caracter, porque en los lenguajes normales tenemos más de un caracter, como `for`.
- $\alpha \in (V_T \cup \{A_i\})*$, donde A
- $\beta \in (V \cup \{\#\})*$

Tenemos un caracter que nos permita indicar donde se acaba el algoritmo, asi como en un archivo hay `EOF`, nosotros tenemos #

El estado inicial es $(q, 1, \varepsilon, S\#)$, donde S es la cadena a analizar y al final viene el simbolo que lo termina, #

Tenemos que estudiar 211, 212, hacerlo en lo que queramos pero siguiendo el algoritmo, quiere el del libro, que traduzcamos el pseudocódigo.

Si la cadena de entrada no es generada por la gramática, entonces tiene que probar todos los casos.

8.19. Flex

1er regla: Si tu autómata está en un estado de avance, tiene una α que ya reconocio el examen y hay un no terminal, lo que hace el automoata es tomar la primer regla y pone toda la parte beta

Pagina 213

La regla 3, si el automata esta en el estado de avance y reconoci un n+1 caracter, si en la parte alpha hay un monton de cosas y en beta hay un gatito, entonces a esto se le llama estado terminal, que quiere decir que la cadena terminal, lo que pasa a el mismo estado con cadena vacia en beta, es decir que ya reconocio todo.

El problema del algoritmo es que hay backtrack, al final queremos que nos de la secuencia sin backtrack.

Eso es el paso 4, el automata, si encuentra algo que no reconoce, automaticamente entra en estado de backtrack b, estando en estado de backtrack, si lo ultimo que hay en alpha es un terminal, pasa a beta e i disminuye en 1.

6. Si estamos en backtrack, en alpha hay un no terminal con su regla j, u en beta tenemos una regla j de Beta, si tenemos mas opciones de reglas entonces ponemos otra de las reglas. Si no hay más reglas, y a es el simbolo inicial, entonces **la cadena no es reconocida**. Si no hay alternativa y no es el simbolo inicial, pasa a dejar el caracter en beta no terminal y sigue en backtrack.

En el cuaderno tengo ejemplos

8.20. Sobre las gramaticas como funciones recursivas

A alguien se le ocurrió que podia ser recursividad, una regla del tipo $S \to aS$, lo que desde el codigo se podría ver:

```
// Y apunta a la cadena de entrada
bool S(int i)
    si entrada[i] == a
        return S(i+1)
    else
        return false
```

Se ocurrio que todo pudiera ser escrito de forma recursiva, y traducian todas las reglas *literalmente* todas las reglas de forma recursiva. El problema es que no todos los lenguajes soportaban este tipo de patrones y agotan rapido la memoria.

8.21. Algoritmo de reconocimiento de Knuth

Pag 231

Inventó un algoritmo de reconocimiento sintáctico, donde escribe la gramática en formas de tablas, la tabla podría reconocer la gramática y no deberíamos de tener que usar backtracking.

Top down es porque comenzamos del simbolo inicial y vamos hacia abajo

Lo único que pide es que sean gramaticas de la forma:

$$X \to Y_1 \mid Y_2 \mid ... \mid Y_m \mid Z_1 Z_2 ... Z_n \quad \text{donde ambos } Y_i \text{ y } Z_i \text{ inician con no terminales o terminales}$$

Este tipo de relgas las pasamos de la forma:

$$X \to Y_1'$$

Tenemos que agregar una $E'' \to TE'$ y $T'' \to *FT'$, esto se debe, si vemos la gramatica en la página La primer regla de E cumple con el requisito, inicia conun no terminal La tercera regla comienza con un terminal, es por eso que lo convertimos a E''

Quedando de la forma:

$$E' \to E''|\varepsilon$$
$$E'' \to TE'$$

Lo mismo sucede con la tercera regla, la de T

La última regla no tiene problemas porque una comiuenza con terminal Y_1 y la (E) es de la forma $Z_1...Z_n$

La regla de la gramática queda:

$$S \to E\#$$
$$E \to TE'$$
$$E' \to E''$$
$$E' \to \varepsilon$$
$$E'' \to +TE'$$
$$T \to FT'$$
$$T' \to T''$$
$$T' \to \varepsilon$$
$$T'' \to *FT'$$
$$F \to b$$
$$F \to (E)$$

En fuerza bruta, para verificar si es verdadero tenemos que ver, en el caso de S, ver que genere una E y después $\#$.

El código operativo para sintetizar a una E (escrito `[E]`), si el codigo operativo responde true o false irá a distintas partes.

En el primer ejemplo, si `[E]` devuelve false entonces es `ERROR`. En cambio si es true se va a el siguiente (se escribe vacío).

Cunado dice false, se quiere decir que el codigo operativo devuelve false a quien lo invoco. Lo mismo cuando dice true. Si está vacío quiere decir que se tiene que realizar la siguiente regla.

Si se trata de una regla que genera ε, si tenemos reglas del tipo $X \to \varepsilon$ hay que crear una fila de la tabla que tenga un no terminal que no exista en la gramática, que siempre devuelve true en error y no error.

```
X    [N]    true    true
N    a      false   false
```

Como no se tiene que leer nada de la cadena de entrada, si tenemos una regla vacia nos inventamos el no terminal que no existe en la gramática y poner el false false. Donde tenemos que poner un terminal de la gramática.

Ejemplo $(b * b)\#$

La ventaja de esto es que puede ser usado

La h es el contador de *matcheados*. Es top-down porque comienza en el inicio. Se les ocurrio que si tenian cierta información de la cadena de entrada, lo tenían que usar.

De aqui surge el concepto de las gramáticas LL(1).
- L: Que en una expresión el símbolo que va primero a extender es el que está más a la izquierda
- L: De la cadena de entrada nos vamos a fijar en el simbolo que no hemos reconocido, ejemplo si reconocimos ya 2 seria 23(3)we, donde 3 es el que debe seguir.
- 1: Que se ve solo uno a la izquierda.

Definición de una gramática LL(1), hay 3 simple, generales y con reglas vacías. No quiere la definición.

- Tiene que ser independiente del contexto
- Es decir, que no tiene que ser tampoco regular
- No puede tener reglas vacías
- Para un no terminal cualquiera, las reglas tienen la forma $A \to a_1\alpha_1 \mid a_2\alpha_2 \mid ... \mid a_n\alpha_n$ Donde $a \in V_T$ y $\alpha \in V^*$, puede ser cualquier secuencia, excepto reglas vacías, porque al frente siempre habrá por enfrente un terminal.

  Cada una de las reglas debe tener un terminal al inicio distinto

Primer ejercicio del examen

Primero probamos que la gramatica es regular.

- La longitud de α debe ser menor a β
- La parte beta tiene que ser de la forma terminal o terminal_noterminal
- Para ver si es independiente del contexto es que en α no hay terminales
- La regla S cumple con que el primero siempre sea diferente y terminal, se cumple tanto en A como en S.

En la página, en cuanto a tabla dice el profesor, en M : las primeras son las filas, es decir en las filas tenemos todos los elementos del vocabulario y el gatito. Y en las colunmnas tenemos Los terminales y el gato. Y en las celdas tenemos

Si estamos en la fila A (indicador de fila, puede ser terminal o no terminal) y columna a (indicador de columna). Si en fila y columna tenemos n cosa podemos mapear que se escribirá de la pagina 239.

Ejemplo:

$$
\begin{aligned}
& a\alpha \\
& 1) S \to aS \\
& 2) S \to bA \\
& 3) A \to d \\
& 4) A \to ccA
\end{aligned}
$$

quedará:

	a	b	c	d	#
S	(aS, 1)	(bA, 2)			
A			(ccA, 4)	(d, 3)	
a	pop				
b		pop			
c			pop		
d				pop	
#					accept

Todo lo demás es un error, lo que está en blanco. En la Tabla 6-1 es un ejemplo donde estamos.

En la segunda tenemos $aS\#$ encontramos una a, que es pop, por lo que quitamos la a de ambos lados. Solo concatenamos en la salida el numero de la regla de las tuplas.

La salida indica las derivaciones que tenemos que hacer para replicarlo, similar a lo que sacaba el de fuerza bruta.

Si cuando estamos haciendo el recorrido y caemos en un espacio vacio es error

Las LL1 simples exigen que inicien con un no terminal.

8.22. Gramaticas LL1 sin reglas vacías (pag 243/abs 259)

Primero define una función `FIRST`(α), es el conjunto omega, donde $\omega \in V_T$ de forma que si partimos de un valor de entrada α, al inicio, va a quedar omega, forma parte de *first*

Ejemplos en el libro abs 262:

```
FIRST(S#) =

S# ⇒ ABe# ⇒ {}


FIRST(c) = c
-- Toca imaginarse todas las derivaciones
-- Tenemos que ver el Gra-mix
```

Una gramatica sin reglas vacías es LL1 si:
- Es independiente del contexto
- Si para todas las reglas en las que hay opciones, la intersección de los *first* es el vacío

Error común: «Para ver si esta gramatica es ll1 sin reglas vacias voy a ver el first de S”" S’ no hay que verlo porque solo tiene una opción. No hay que ver el first de donde solo hay una regla. En el caso de la gramatica tenemos que hacerle FIRST a A y B.

Aqui hablo de la gramatica en abs 259 y 243

Solo si el no terminal tiene más de una opción. Para comprobar si se trata de una gramatica tenemos que obtener el first de las dinstintas reglas y después hacer la intersección.

By applying this definition to the sample grammar, we obtain the following:

1. `Rules A → dBlaSIc FIRST(dB) n FIRST(aS)= {d}n <a! = d FIRST(dB) n FIRST(c) = {dIn {c! = ϕ FIRST(aS) n FIRST(c) = (a} n {c) = $ 2. Rules B → ASIb FIRST(AS) n FIRST(b) = {a, c, d}n {b} =ϕ`

Como en todas las reglas las intersecciones de los FIRST resulta en el conjunto vaci

Clave: Los first de la parte derecha de los no terminales que tienen más de una opción

Igual que en LL1 simple se hace una tabla, ejemplo en 247.

En Sa tenemos ABe 1 porque a está en `FIRST` de S, lo mismo para c y d

Para demostrar si es ll1 solo tenemos que ver los first de solo los que tienen màs de una opción, pero para hacer la tabla si necesitamos el first de todos.

El first de A es {a, c, d}

La ejecución es igual. Recordemos que si caemos en un espacio en blanco es un error. Ejemplo en 247, abs 264.

8.23. Gramaticas LL1 con reglas vacías

Este tipo de gramaticas la usa para ejecutar comandos de usuarios, por ejemplo, en bases de datos. De esta forma, si sabe que el query es apropiado, saber si dará vacío.

En este tipo de gramaticas si tenemos reglas vacías. Si vemos los first de la parte derecha de la regla, cumple:
- Es independiente del contexto

Si hacemos la tabla, vamos a encontrar que, si en nuestra cadena de entrada hay une flecha, entonces tendriamos que decir que en la entrada hay un corchete o un punto. Pero si sustituimos la B por la regla vamos a llegar al signo que está en la entrada.

Para solucionarlo no solo hay que calcular el FIRST de las partes derechas, sino también la función `FOLLOW` de los no terminales.

`FOLLOW` se aplica solo a no ternimales, a las partes izquierdas, es el conjunto de terminales ($\omega \in V_T$) de forma que si comenzamos en el símbolo distinguido de la gramática y hacemos muchas derivaciones (o ninguna, por eso =>*), y llegamos a $\alpha, \gamma \in V*$, entonces los ωs son los que están en el FIRST de γ. Si γ es ε, como no hay nada después vale nada.

Los no terminales que tienen más de una regla tienen que cumplir la misma regla de los simples, que su intersección sea $\emptyset$.

Con la gramática de la pag 250.

O. S' -> A# (solo tiene una regla)
1. A - iB ← e (solo tiene una)
2. B -› SB
   - First de SB = [, .
   - Follow B = Siempre habrá ←
3. B -› ε
4. S -> [eC]
   - First [ec] = [
   - No follow porque no tiene reglas vacias
5. S - .i
   - First .i = .
   - No hay follow porque no tiene reglas vacias
6. C -> eC
   - First eC = e
   - Como tiene regla vacia hacemos follow
   - Follow C = Porque su unica sustitución posible lleva a

]
7. C-> ε

Como el follow solo se calcula para no terminales que tienen reglas vacías, lo especñifica en la función $M(A, a)$

En A con i aparece la regla porque en el first está la i. En B ← aparece vacio 3 porque en el follow de B está la flecha a la izquierda.

La ventaja es que se puede automatizar, porque podemos hacer la *gramix*, que es infinita, el conjunto de first y folllows no cambia. Se puede hacer un codigo que le das la gramatica ll1 y genera la tabla.

Estos 6 metodos, fuerza bruta, knuth, ll y demás son sencillos y requieren poco computo. Ahora vamos a ver down top que van sustituyendo a la derecha hasta que queda el sìmbolo inicial, lo que verifica que es generada. , lo que verifica que es generada.

Yacc es down top, lo malo es que requieren más recursos de cómputo.

Los lenguajes que usamos son dependientes del contexto, interpretarlos depende del mismo.

Recursividad por la izquierda el unico que no puede es el de fuerza bruta.

LL1 simple: Cada regla comienza con un no terminal distinto LL1: Donde se usa FIRST LL1 con reglas vacias

8.24.

Ejercicio 8
- Demostrar que es una gramatica LL1
- Ver las intersecciones

- Es independiente del contexto
- Si para todas las reglas en las que hay opciones, la intersección de los *first* es el vacío

```
S' → S#
S → aABC
A → a|bbD
B → a|
C → b|
D → c|

Primero vemos si es regular:
    - En todas las reglas |a| < |b|, se cumple (vacio cuenta como 1)
    - No es regular por la regla 1, 2, 4
Independiente del contexto?
    - Ya sabemos que a < b
    - a solo pertenece a los no terminales, por lo tanto es independiente del contexto
Las reglas que tienen mas de una opcion:
    Por culpa de A neceistamos FIRST(a) y FIRST(bbD)
        - First(a) = a
        - First(bbD) = b   No se interseccciona
    Por culpa de B vemos el FIRST(a) y first(empty)
        Como es empty queda FOLLOW(B) =
        El follow de B es el first de C
            → FIRST(C) = b
        + FIRST(a) = a
    Por culpa de C vemos que FIRST(b) y FOLLOW(C)
        El FIRST de b = b
        El FOLLOW de C = \#
    Por culpa de D vemos FIRST(c) = c
        FOLLOW(D) = quedara como bbDCBC, por lo que veremos el first de BC
        FIRST(BC) = {a, b}
        como BC primero sacamos el first de b, que es a.
        Pero como en BC, B puede ser vacio, entonces necesitamos tambien obtener el first de
C, que es b
        - Haciendo el gramix, la unica forma en que D entra en las derivaciones es que S
pase a ser donde A y A pase a ser abbD seguido de BC donde BC es lo que le sigue, por eso
sacamos el first de todo eso
        - Como B puede ser vacio tambien tenemos que sacar el first del que le sigue, D
        - En el gramix analizamos todos los patrones que pueden generar a D, en este caso,
no es solo uno
        {a, b} sect { c } = emptyset
```

En la tabla no se considera el vacio

Dejar espacio suficiente para que vea Llenamos con

- Los first
- Para los que tienen reglas vacias con los follow

– estos first van a parte S': FIRST(S#) = a – no lo quitemos en el examen S: FIRST(aABC) = a Reusamos el FIRST DE A en a o bbD Reusamps F(B) = a

No quiere explicacion de las triviales Como D tiene vacio tambien asignamos todos los follows

- a, b son ambos el numero 10

```
    a          b        c        d         #
S'  (S#, 1)
S   (aABC,2)
A   (a, 3)     (bbD, 4),
B   (a, 5)     (epsilon, 6), ,               (epsilon, n)
C               (c, 7)                       (epsilon, 8)
D   (epsilon, 10), (epsilon, 10), , (c, 9),  (epsilon, n)

a
b
c
#
```

Y ahora probamos que una cadena se genera con esa gramatica:

El libro no tiene ejemplos pero podemos inventarnos ejemplos
- (abbc, S#, emptyset)
- en S con a tenemos aABC 2
- (abbc, aABC#, 2)
- pop
- (bbc, ABC, 2)
- A con b tiene bbD
- (bbc, bbDC, 4)
- pop 2 veces
- (c, DC, 4)

Se equivoco en los follow porque en el Follow de C le falto ver que pasa si esta vacio, es decir que es # entonces C lleva a gato, D lleva a gato

Eventualmente lleva al mero mero

```
S'  &→ S#
    - Solo tiene una regla, no revisamos
S  &→ aABC
    - Solo tiene una regla, no revisamos
A  &→ a | bbD
    - Hay dos reglas
    - FIRST(a) =? FIRST(bbD)
    - a sect bbD = emptyset se cumple
B &→ a | epsilon
    - Tiene regla vacia
C &→ b | epsilon
    - Tiene regla vacia
D  &→ c | epsilon
    - Tiene regla vacia
```

Es independiente del contexto porque no tenemos terminales en la parte izquierda de las reglas.

9. Expresiones aritmeticas

Pagina 277

- Infija: Operando Op Operando
- Prefija* (polaca): Op Operando Operando
- Post-fija* (polaca inversa): Operando Operando Op

*: No hace falta tener paréntesis. Se les llama expresión polaca y polaca inversa.

Como la sufija y post-fija no requieren paréntesis, se puede usar con pilas las operaciones. Por eso se pueden convertir las expresiones en uno de estos formatos.

9.1. Convertir de infija a polaca

Usando la tabla de precedencia de la pag 279 (abs 296), se puede mapear cada elemento de la expresión. Hay que tomar en cuenta que el algoritmo quita los enteros negativos pues es considerado como operador.

Entonces mete valores. Checa la precedencia, si es menor se quitan los elementos del stack hasta que se encuentre uno mayor.

En el caso de que existan paréntesis, existe f y g, a diferencia de solo f. Asume que los identificadores son de un solo caracter.

En un compilador completo no se usa en un compilador completo, porque la misma gramática se encarga de indicar la sintaxis correcta. Pero en el libro tenemos un algoritmo pequeño para verificarlo.

9.2. Gramatica de operadores

Aquella que no tiene reglas del tipo $V \to \alpha XY\beta$ donde $X, Y \in V_N$. Es decir, que no aparecen no terminales consecutivamente. Es decir, que siempre entre no terminales habrá un terminal.

En el caso de las funciones como $\sin(x)$ se tiene que agregar de la forma:

$$\begin{matrix} \dots \\ P \to i \mid (E) \mid \sin(E) \end{matrix}$$

Las expresiones algebraicas se pueden traducir en forma de tablas.

7.2 Podemos definir tres tipos de procedencias relacionales:

- Son iguales, si existe una producción $U \to aSSB$ o $U \to aSXSB$. En el segundo caso están separados por un no terminal.
- Es menor si tiene $U \to \alpha S_1 X\beta$ tal que $X \Rightarrow +S_2 \dots$, X con por lo menos una derivacion deriva en S_2 al inicio.
- Es mayor si existe $\alpha X S_2 \beta$ donde $X \Rightarrow +$ en algo que al final tiene a S_1 o $X \Rightarrow +\dots S_1 Y$

$$E \to T|E+T|E-TT-F|T*F|\frac{T}{F}F \to P|F\hat{}PP \to i|(E)$$

En la gramatica, en la expresion tenemos un a que es vacio S1 que es (el no terminal y otro S2). Una producción es la gramix, podemos partir de donde sea.

En la tabla que decribe la gramática de arriba se aprecia la precedencia para todos los caracteres.

- Ejemplo $(< ($. Necesitamos encontrar una producción que hace que se derive en (al inicio

```
s1  x  s2
(   E  )
```

E => T => F => P equiv ((

Al final nos quedará:

```
(( ...
```

Ojo que `...` pueden ser ε

Por eso es que es menor

- Ejemplo $i > +$

Para que se cumpla necesitamos una produccion que diga alpha x s2

```
Parte de
X  s2  beta
T  *   F

Vamos a ver si llegamos por derivaciones a un operador
T * F ⇒ F * F ⇒
```

También podemos notar que, en el caso de el `*`, el único que rompe la

```
S1 > S2
/ y +

Parte de
T / F

X s2
T /  F

Por derivaciones de T queremos llegar a uno que tenga

T / F
```

⇒ F / F
⇒ P / F

La relacion que queremos ver es S1 := / y S2 := +

    s2
E   +   T
⇒ T + T
⇒ T / F + T

Y ese es el caso de

Con la tabla de precedencia se toma una empresión y se colocan las relaciones entre cada operador, u camos sustituyendo cada uno por reglas

De esta forma vamos desde abajo, haciendo sustituciones hasta que llegamos al simbolo inicial.

Se entrega un día antes el proyecto final:
- Compilador de tipo traductor que recibe codigo en pascal y lo traduce a C
- El chiste es que no necesitamos saber pascal porque ya tenemos la gramatica (bkf, backus normal form)
- Se entrega un documento y todo el codigo
- La documentación explica absolutamente todo
- El proyecto es traducir de Pascal a C/C++: Recibe un programa en pascal y devuelve su equivalente en C
- La gramatica tiene no terminales con más de una letra. En el documento que nos va a dar viene en negritas los no terminales. El no terminal inicial es programs
- Mes y medio queda
- No todo debe determinarse desde bison, por ejemplo un identificador puede usarse una expresión regular
- Cuando encontremos identificadores tenemos que verificar que no sean palabras reservadas

 - Los comentarios asumimos que son iguales que en C/C++, con `\/\/` y `\/* ... */`

Consejos:
- Advierte que necesitamos estudiar como se hace un arbol, porque eventualmente vamos a hacer un arbol de sintaxis
- Tambien nos recomienda que hagamos una tabla de *hashes*.
- No esperarnos
- Para encontrar un trabajo final asi deberíamos buscar 12 años atrás.
- Revisa GitHub para ver que no hay nada similar
- Todo el código es C o C++

9.3. Bison

Parte de un archivo de entrada donde escribimos con un formato específico de `yacc`. En él escribimos la gramática. Trataron que fuera lo más similar a `flex`.

Lo único que pide es que la gramática sea independiente del contexto Va entre apostrofos, no entre comillas

$$E \to T|E+T|E-T$$
$$T \to F|T*F|\frac{T}{F}$$
$$F \to P|F\hat{}P$$
$$P \to i(E)$$

i + * = >

Lo que hace el algoritmo es buscar de izquierda a derecha una regla o pedazo de la expresión que esté entre mayor y menos, para lo que queda encerrado, debe existir una regla en la gramatica de la cual la parte derecha es i. Con la tabla 7-7 y tabla 8-8. Se sustituye desde el lado izquierdo.

Con `#E#` se acaba

En el examen hay una pregunta en el que pide que hagamos cambios al de fuerza bruta.

9.4. Tablas de símbolos

Contiene:

- Identificador
- De que tipo es: var, procedimiento, funcion, arreglo
- Tipo de variable si es una variable
- Procedimiento: No devuelve nada
- Funcion: Que tipo devuelve la funcion
- Dimensión del arreglo, el tipo si es un arreglo
- El ámbito (scope): Espacio del tiempo de vida de las variables
  - El ambito se va multiplicando x2, iniciando por 1, si tenemos ambito de 8 sabemos que estamos 4 veces dentro
  - Es x2 porque para obtenerlo podemos hacerle shift al bit.
- Direccion de memoria: Donde estan en el heap
- Tamaño que ocupa cada elemento
- Numero de linea donde se define
- Numeros de linea donde se usa/invoca (p. ej. no podemos tener una variable que no se definio antes)
- Tenemos que hacer una tabla de hashes

9.5. Tabla hash

Ocupa una función hash que mapea de una cadena de caracteres a numeros naturales $t_1t_2t_3...t_i \to \mathbb{Z}$

Colisión: cuando hay dos elementos distintos que mapean al mismo valor.

Sumar los codigos ascii de la cadena

Un ejemplo de tabla es que define un arreglo de N elementos. Para cada valor que se va a ingresar lo pasa a alguno de los espacios del vector, cuando hay colision lo agrega tambien, porque el arreglo en realidad contiene todos los elementos

- Variable no declarada
- Variable no usada
- Variable declarada dos veces en el mismo ambito

9.5.1. Codigo

```
enum VarType // Tipo de dato y eso
enum VarConcept // Si es arreglo y eso
```

Por cada elemento tenemos
- Tamaño en Bytes
- La linea donde se define
- El ambito donde esta
- Los punteros los tiene porque no quiere crear de golpe los items, hasta que es necesario crea los valores.
- La lista de colisión es un arerglo de vectores
- Para crear un item:
  - Tiene el nombre del identificador
  - Un puntero a item en la memoria
- La tabla tiene una variable count, contando tanto en items como en colision
- Primero crea el item y luego el espacio en la cadena, para destruirlo lo hace en modo inverso
- Para donde puede evitar ifs lo hace
- Si encuentra una colisión, lo pone al final del arreglo
- Cuidado de tener copias no profundas
- Usa unsigned int

- Hay dos casos:
  - Es null (Es el primero)
  - Si no lo agrega a la lista de colisión
- Tiene un metodo de busca que nos permite buscar cierto elemento en la tabla
- BUscar:
  - Busca el indice y pide el item en esa posicion, si no es nulo devuelve que si lo encontro
  - Si es falso puede que el elemento no este y busca uno a uno en la lista de colision
  - Devuelve en que posicion del vector de colisiones esta
- Borrar un elemento:
  - Tiene tabla y llave
  - Si no existe devuelve
  - Si no tiene lista de colision en esa posicion es un error
  - Esta en el lugar y no tiene lista de colision, entonces lo borra directamente
  - Ese indice coincide con el elemento, pero tiene lista de colision, por eso tiene que borrar el elemento e ir copiando los elementos para “comprimir”
- Buscar un elemento en la tabla y devuelve el index, tanto de la lista de colision como el de los items
- Un metodo para imprimir la tabla completa
- Tiene un arreglo de items y un arreglo de colision

Temas del examen
- Gramaticas con atributos
- Reconocerodes SSLR1
- Arbol sintactico
- Tabla de simbolos
- Manejo de errores

Para juntarlo todo:
- Cuando encontremos un identificador, lo subimos a la tabla de simbolos, mandando a la vez las lineas y scope
- Obviamente hay que agregar los distintos tipos, como program si no vienen
- Variables, nombres de procedimientos, funciones, etc. Toca buscar exactamente donde los vamos a encontrar