Selfa Pro HowTo

Definiciones

Expresiones Regulares

regexp *nombre_expresión* **{** *expresión* **}**

expresión: Se forma con letras mayúsculas y minúsculas, cifras, cadenas entre comillas simples (') y los símbolos $ y @ que representan la cadena vacía y el lenguaje vacío respectivamente.

*Sobre estos se pueden aplicar las operaciones | (intersección), [] (clase, separando dos elementos por un guión), concatenación (operador implícito), * (clausura), + (clausura positiva) , y ? (opcionalidad)*

Ejemplos:

- **regexp** e1 { (abc)| ($| 'hola') }
- **regexp** e2 { @* }
- **regexp** e3 { ([A-C]*)(b?) }

Consejo: Se recomienda siempre el uso de paréntesis para obtener las preferencias adecuadas.

Gramáticas

grammar *nombre_gramática* **{**
terminal *terminales_separados_por_comas* **;**
nonterminal *no_terminales_separados_por_comas* **;**
axiom *no_terminal_simbolo_inicial* **;**
productions {
no_terminal **:=** *partes_derechas* **;**
…
no_terminal **:=** *partes_derechas* **;**
}
}

partes_derechas: Son las partes derechas de la producción separadas por el símbolo |. Si la parte derecha es vacía se usará $.

Ejemplo:

```
grammar gram {
        terminal a,b ;
        nonterminal A,C ;
        axiom A ;
        productions {
                A := b b | a a | a b | a C b ;
                C := a a| b b | $ ;
        }
}
```

Autómatas

automaton *nombre_automata* **{**
states *estados_separados_por_comas* **;**
alphabet *alfabeto_separado_por_comas* **;**
initial *estado_inicial* **;**
final *estados_finales_separados_por_comas* **;**
transition {
estado **,** *alfabeto_o_$* **=** *destinos* **;**
…
estado **,** *alfabeto_o_$* **=** *destinos* **;**
}
}

destinos: Destinos son los estados a los que se llega. Puede ser uno solo o varios entre paréntesis separados por comas.

Ejemplo:

```
automaton prueba {
        states q0,q1,q2,q3;
        alphabet a,b;
        initial q0;
        final q3;
        transition {
                q0,$=(q1,q3);
                q1,a=q1;
                q1,b=q2;
                q2,$=q3;
        }
}
```

Autómatas a Pila

pdautomaton *nombre_automata_pila* **{**
states *estados_separados_por_comas* **;**
alphabet *alfabeto_separado_por_comas* **;**
stack *alfabeto_pila_separado_por_comas* **;**
initial *estado_inicial* **;**
stackinitial *elemento_inicial_pila* **;**
[final *estados_finales_separados_por_comas* **;]**
transition {
estado **,** *alfabeto_o_$* **, (** *alfabeto_pila_o_$* **) =**
estado **, (** *nuevos_elementos_pila* **);**
…
}
}

nuevos_elementos_pila: Son los elementos de la pila que se introducen, sustituyen a alfabeto_pila_o_$. Van separados por comas. Si no hay ninguno que meter se usa $. El primero será el que quede en el tope de la pila.

Ejemplo:

```
pdautomaton prueba {
        states q0,q1,q3;
        alphabet a,b;
        stack A,B,Z0;
        initial q0;
        stackinitial Z0;
        transition {
                q0,a,(Z0)=q0,(A,Z0);
                q0,a,(A)=q0,(A,A);
                q0,b,(A)=q1,($);
                 q1,b,(A)=q1,($);
                q1,$,(Z0)=q1,($);
        }
}
```

Selfa Pro HowTo

Operaciones

Expresiones Regulares

alternation: Hacer la unión de dos expresiones regulares
*Sintaxis: expresion=**alternation**(expresion_a,expresión_b);*

concatenation: Hacer la concatenación de dos expresiones.
*Sintaxis: expresion=**concatenation**(expresion_a,expresión_b);*

equals: Comprobar que dos expresiones son equivalentes.
*Sintaxis: **equals**(expresión_a,expresión_b);*

print: Mostrar por pantalla una expresión regular.
*Sintaxis: **print**(expresión_regular);*

recognize: Saber si determinada entrada de símbolos es reconocida por una expresión regular.
*Sintaxis: **recognize**(expresión, símbolos_separados_por_ espacios);*

REtoFA: Obtener el autómata finito con lambda transiciones equivalente a la expresión usando el algoritmo de Thompson.
*Sintaxis: autómata=**REtoFA**(expresión_regular);*

star: Hacer la clausura de Kleene a una expresión regular
*Sintaxis: expresion=**star**(expresion);*

Autómatas

complement: Realizar el complemento de un autómata
Sintaxis: autómata_complemento=**complement**(autómata);

equals: Comprobar que dos autómatas son equivalentes.
*Sintaxis: **equals**(autómata_a,autómata_b);*

FAtoDFA: Pasar un Autómata Finito No Determinista con lambda transiciones a determinista.
*Sintaxis: autómata_determinista=**FAtoDFA**(autómata);*

FAtoMDFA: Pasar un Autómata Finito No Determinista conlambda transiciones a AFD Mínimo.
*Sintaxis: autómata_mínimo=**FAtoMDFA**(autómata);*

FAtoNDFA: Pasar un Autómata Finito No Determinista conlambda transiciones a AF no determinista.
*Sintaxis: automata_sin_lambda=**FAtoNDFA**(autómata);*

FAtoRE: Obtener la expresión regular equivalente al autómata de entrada.
*Sintaxis: expresión_regular=**FAtoRE**(autómata);*

FAtoRG: Obtener la gramatica regular equivalente a un automata.
*Sintaxis: gramática_regular=**FAtoRG**(autómata);*

intersection: Realizar la interseccion de dos autómatas.
*Sintaxis: autómata_intersección=**intersection**(auta,autb);*

inverse: Realizar el inverso de un automata.
*Sintaxis: autómata_inverso=**inverse**(autómata);*

print: Mostrar por pantalla un autómata.
*Sintaxis: **print**(autómata);*

union: Realizar la union de dos autómata.
*Sintaxis: autómata_unión=**union**(auta,autb);*

recognize: Reconocer una cadena con un autómata.
*Sintaxis: **recognize**(autómata, símbolos_separados_por_ espacios);*

visualrecognize: Reconocer una cadena con de manera visual.
*Sintaxis: **visualrecognize**(automata,símbolos_separados_por_ espacios);*

Gramáticas

CFGtoPDA: Obtener un autómata a pila equivalente a una gramática libre de contexto:
*Sintaxis: autómata_pila=**CFGtoPDA**(gramática_libre_contexto);*

clean: Realizar la limpieza de una gramática.
*Sintaxis: gramática_limpia=**clean**(gramática);*

cyk: Algoritmo CYK para gramáticas. Sirve para hacer un recognize (ver expresiones regulares o autómatas) sobre una gramática.
*Sintaxis: **cyk**(gramática, símbolos_separados_por espacios);*

equals: Comprobar que dos gramáticas son equivalentes.
*Sintaxis**: equals**(gramática_a,gramática_b);*

fnc: Algoritmo FNC para gramáticas.
*Sintaxis: gramática_fnc=**fnc**(gramática);*

fng: Algoritmo FNG para gramáticas.
*Sintaxis: gramática_fng=**fng**(gramática);*

nullable: Obtencion de no terminales que derivan en la palabra vacia
*Sintaxis: **nullable**(gramática);*

print: Mostrar por pantalla una gramática.
*Sintaxis: **print**(gramática);*

reminacc: Eliminación de símbolos inaccesibles de una gramática
*Sintaxis: gramática_sin_inaccesibles=**reminacc**(gramática);*

remnongen: Eliminación de producciones que no derivan encadena de terminales de una gramática
*Sintaxis: gramática_sin_no_generativas=**remnongen**(gramática);*

remnullable: Eliminación de la palabra vacia
*Sintaxis: gramática_sin_lambda=**remnullable**(gramática);*

remunit: Eliminación de producciones unitarias
*Sintaxis: gramática_sin_unitarias=**remunit**(gramática);*

remuseless: Eliminación de símbolos inútiles de una gramática
*Sintaxis: gramática_sin_inutiles=**remuseless**(gramática);*

RGtoFA: Obtener el autómata finito equivalente a una gramática regular.
*Sintaxis: autómata=**RGtoFA**(gramática);*

Autómatas a Pila

print: Mostrar por pantalla una autómata a pila.
*Sintaxis: **print**(autómata_pila);*

recognize: Reconocer una cadena con un autómata a pila.
*Sintaxis: **recognize**(autómata_pila, símbolos_separados_por_ espacios);*

PDAtoCFG: Obtener la gramática libre de contexto equivalente a un autómata a pila.
*Sintaxis: gramática_libre_contexto**=PDAtoCFG**(autómata_a_pila);*

PDAtoFPDA: Obtener un autómata a pila que trabaje bajo el criterio de los estados finales.
*Sintaxis: automata_pila_finales**=PDAtoFPDA** (autómata_a_pila);*

PDAtoEPDA: Obtener aut a pila que trabaje bajo el criterio de pila vacía.
*Sintaxis: automata_pila_vacía**=PDAtoEPDA** (autómata_a_pila);*

visualrecognize: Reconocimiento de una serie de símbolos por un autómata a pila de manera visual.
*Sintaxis: **visualrecognize**(automata_pila_determinista, símbolos_ separados_por_espacios);*