Paradigmas de la Programación
Laboratorio 1: Sintaxis y Semántica

Laura Alonso Alemany Ezequiel Orbe

En este laboratorio vamos a introducirnos al análisis léxico y sintáctico mendiante el desarrollo de un parser utilizando herramientas específicas para dicha tarea, como son `Flex`, `Bison` y `C`.

Nuestro objetivo será desarrollar un evaluador de fórmulas proposicionales que nos permita evaluar si una determinada fórmula proposicional es cierta o falsa, dada una asignación para sus variables.

Para ello deberemos desarrollar:

- El **parser de fórmulas proposicionales**, el cual dada una fórmula válida deberá retornar su correspondiente árbol sintáctico.
- El **parser de asignaciones**, el cual deberá retornar un diccionario conteniendo las asignaciones para cada variable proposicional.
- El **evaluador de fórmulas proposicionales**, el cual dada una fórmula y una asignación determina el valor de verdad de la fórmula (si la fórmula es verdadera o falsa).

Características de la presentación

- El trabajo es en grupo: máximo: **dos integrantes**, mínimo: **dos integrantes**.
- Fecha de Entrega: Hasta las 23:59:59 del 26/03/2015.
- Formato de entrega:
  1. Empaquetar el directorio `prop-eval` en un archivo llamado `<dni>-lab-1.tar.gz`, donde `<dni>` es el DNI de alguno de los integrantes del grupo.
  2. Enviar el archivo `<dni>-lab-1.tar.gz` por mail a la dirección:

     `paradigmas@famaf.unc.edu.ar`

     El título del mail debe decir:`lab-1`.
  3. Los trabajos enviados fuera de término o que no cumplan con las condiciones de presentación no serán considerados.

Software Requerido

Para realizar este laboratorio necesitaremos el siguiente software:

- Compilador de C
- Bison
- Flex

Las tres herramientas usualmente estan presentes en cualquier distribución estándar de Linux. En Windows podés instalarlas, instalando `gnuwin32` (`http://gnuwin32.sourceforge.net/`).

Código Inicial

Para realizar este laboratorio, utilizaremos el código disponible en:

`http://cs.famaf.unc.edu.ar/materias/paradigmas/sites/default/files/labs/lab-1.tgz`

Al extraer el contenido del archivo, tenemos la siguiente estructura de directorios:

```
prop-eval
├── src
│   ├── aparser ..... Parser de asignaciones
│   ├── fparser ..... Parser de fórmulas
│   ├── eval ..... Evaluador de fórmulas
│   ├── data ..... Estructuras de datos
│   ├── utils ..... Rutinas comunes
│   ├── prop-eval.c ..... Ejecutable
│   └── makefile
└── tests
    ├── test-fórmulas ..... Tests Sección1
    ├── test-assignments ..... Tests Sección2
    └── test-evaluator ..... Tests Sección3
```

1 Parser de Fórmulas

Iniciaremos el desarrollo por el parser de fórmulas. Al finalizar esta sección deberemos tener un parser de fórmulas completo.

1.1 Gramática Inicial

A continuación se presenta la BNF que describe **parcialmente** las fórmulas que utilizaremos:

⟨*forms*⟩ ::= '`begin`' ⟨*phi*⟩ '`end`'

⟨*phi*⟩ ::= ⟨*prop*⟩
| ⟨*phi*⟩ '`v`' ⟨*phi*⟩
| ⟨*phi*⟩ '`&`' ⟨*phi*⟩

⟨*prop*⟩ ::= '`P`'⟨*id*⟩

⟨*id*⟩ ::= '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`'
| '`1`'⟨*num*⟩ | '`2`'⟨*num*⟩ | '`3`'⟨*num*⟩ | '`4`'⟨*num*⟩ | '`5`'⟨*num*⟩
| '`6`'⟨*num*⟩ | '`7`'⟨*num*⟩ | '`8`'⟨*num*⟩ | '`9`'⟨*num*⟩

⟨*num*⟩ ::= '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`' | '`0`'
| '`1`'⟨*num*⟩ | '`2`'⟨*num*⟩ | '`3`'⟨*num*⟩ | '`4`'⟨*num*⟩ | '`5`'⟨*num*⟩
| '`6`'⟨*num*⟩ | '`7`'⟨*num*⟩ | '`8`'⟨*num*⟩ | '`9`'⟨*num*⟩ | '`0`'⟨*num*⟩

1.1.1 Consignas

1. Completá el parser y el lexer de fórmulas que se encuentran **parcialmente** implementados en los archivos `fparser/fparser.y` y `fparser/fparser.l`, para que se puedan parsear todas las fórmulas descriptas por la BNF.

2. La gramática descripta por la BNF ¿es ambigua? Si tu respuesta es afirmativa, modificá el parser para que no sea ambiguo.

3. Queremos que nuestras fórmulas no tengan que estar necesariamente en mayúsculas (ej.: `P1 v P2`) sino que mezclen mayúsculas y minúsculas (ej.: `p1 v P2`). Modificá el parser para que ambas fórmulas sean válidas.

4. Utilizá los tests unitarios (ejecutando `test-fórmulas/make test1`) para verificar que los avances no producen errores.

   Importante: que no existan errores en los tests unitarios no significa que la implementación sea correcta. Es altamente probable que los tests no sean exhaustivos, por lo cual si se te ocurre una nueva prueba no dudes en agregarla al test. Cuando evaluemos tu código agregaremos tests que no estan incluidos en esta distribución.

1.2 Extendiendo la Gramática.

Ahora vamos a extender la gramática del punto anterior para que podamos utilizar otros conectivos lógicos.

⟨*forms*⟩ ::= '`begin`' ⟨*phi*⟩ '`end`'

⟨*phi*⟩ ::= ⟨*prop*⟩
| ⟨*phi*⟩ '`v`' ⟨*phi*⟩
| ⟨*phi*⟩ '`|`' ⟨*phi*⟩
| ⟨*phi*⟩ '`&`' ⟨*phi*⟩
| ⟨*phi*⟩ '`^`' ⟨*phi*⟩
| ⟨*phi*⟩ '`;`' ⟨*phi*⟩
| ⟨*phi*⟩ '`-->`' ⟨*phi*⟩
| ⟨*phi*⟩ '`->`' ⟨*phi*⟩
| ⟨*phi*⟩ '`<-->`' ⟨*phi*⟩
| '`!`'⟨*phi*⟩
| '`-`'⟨*phi*⟩
| '`~`'⟨*phi*⟩
| '`(`'⟨*phi*⟩'`)`'
| '`true`'
| '`false`'

⟨*prop*⟩ ::= '`P`'⟨*id*⟩

⟨*id*⟩ ::= '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`'
| '`1`'⟨*num*⟩ | '`2`'⟨*num*⟩ | '`3`'⟨*num*⟩ | '`4`'⟨*num*⟩ | '`5`'⟨*num*⟩
| '`6`'⟨*num*⟩ | '`7`'⟨*num*⟩ | '`8`'⟨*num*⟩ | '`9`'⟨*num*⟩

⟨*num*⟩ ::= '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`' | '`0`'
| '`1`'⟨*num*⟩ | '`2`'⟨*num*⟩ | '`3`'⟨*num*⟩ | '`4`'⟨*num*⟩ | '`5`'⟨*num*⟩
| '`6`'⟨*num*⟩ | '`7`'⟨*num*⟩ | '`8`'⟨*num*⟩ | '`9`'⟨*num*⟩ | '`0`'⟨*num*⟩

1.2.1 Consignas

1. Extendé el parser y el lexer para que se puedan parsear todas las fórmulas descriptas por la BNF, teniendo en cuenta lo siguiente:
   - '`v`' y '`|`' representan la `disjunción`.
   - '`&`','`^`' y '`;`' representan la `conjunción`.
   - '`-->`' y '`->`' representan la `implicación`.
   - '`!`','`-`' y '∼' representan la `negación`.
2. Asegurate de que la gramática resultante no sea ambigua.
3. Utilizá los tests unitarios (ejecutando `test-fórmulas/make test2`) para verificar los avances.

2 Parser de Asignaciones

Ahora desarrollaremos el parser de asignaciones. Al finalizar esta sección deberemos tener un parser de asignaciones completo, que genere una estructura de datos con las asignaciones para cada variable.

2.1 Gramática.

A continuación detallamos la BNF que describe parcialmentes las asignaciones:

⟨*asgmnt*⟩ ::= ⟨*vval*⟩
| ⟨*vval*⟩‘`,`’⟨*asgmnt*⟩
| ⟨*vval*⟩‘`\n`’⟨*asgmnt*⟩

⟨*vval*⟩ ::= ⟨*prop*⟩‘`:`’⟨*bool*⟩

⟨*prop*⟩ ::= ‘`P`’⟨*id*⟩

⟨*id*⟩ ::= ‘`1`’ | ‘`2`’ | ‘`3`’ | ‘`4`’ | ‘`5`’ | ‘`6`’ | ‘`7`’ | ‘`8`’ | ‘`9`’
| ‘`1`’⟨*num*⟩ | ‘`2`’⟨*num*⟩ | ‘`3`’⟨*num*⟩ | ‘`4`’⟨*num*⟩ | ‘`5`’⟨*num*⟩
| ‘`6`’⟨*num*⟩ | ‘`7`’⟨*num*⟩ | ‘`8`’⟨*num*⟩ | ‘`9`’⟨*num*⟩

⟨*num*⟩ ::= ‘`1`’ | ‘`2`’ | ‘`3`’ | ‘`4`’ | ‘`5`’ | ‘`6`’ | ‘`7`’ | ‘`8`’ | ‘`9`’ | ‘`0`’
| ‘`1`’⟨*num*⟩ | ‘`2`’⟨*num*⟩ | ‘`3`’⟨*num*⟩ | ‘`4`’⟨*num*⟩ | ‘`5`’⟨*num*⟩
| ‘`6`’⟨*num*⟩ | ‘`7`’⟨*num*⟩ | ‘`8`’⟨*num*⟩ | ‘`9`’⟨*num*⟩ | ‘`0`’⟨*num*⟩

⟨*bool*⟩ ::= ⟨*true*⟩ | ⟨*false*⟩

⟨*true*⟩ ::= ‘`true`’ | ‘`1`’

⟨*false*⟩ ::= ‘`false`’ | ‘`0`’

2.1.1 Consignas

1. Implementá el lexer partiendo del archivo `aparser/aparser.l`.
2. Implementá el parser partiendo del archivo `aparser/aparser.y`. El parser debe generar una estructura de datos como la de `assignment.h`, con las asignaciones para cada variable.
3. Utilizá los tests unitarios (ejecutando `test-assignments/make`) para verificar que los avances no den errores.

3 Evaluador de Fórmulas

A continuación desarrollaremos el evaluador de fórmulas proposicionales. Este evaluador evalúa si una fórmula (analizada con el parser de fórmulas) es cierta, dada una asignación de valores para las variables (obtenida mediante el parser de asignaciones).

3.1 Consignas

1. Implementá la función `eval` en el archivo `evaluator.c`
2. Completá la función `main` en el archivo `prop-eval.c`
3. Utilizá los tests unitarios (ejecutando `test-evaluator/make`) para verificar que los avances no den errores.