Universidad ORT Uruguay
Facultad de Ingeniería

InterfLan 1.0: Interface Language

Lenguaje de especificación de interfaces para ingreso y validación de datos

Entregado como requisito para la obtención del título de Ingeniero en Sistemas

Emil Santurio - 180754

Tutor: Álvaro Tasistro

2018

Declaración de autoría

Yo, Emil Santurio, declaro que el trabajo que se presenta en esta obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba el Proyecto;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mí;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Emil Santurio
1-3-2018

“A mi madre, que siempre por lo que sea me apoya a alcanzar todos mis objetivos. A Tato, más que un tutor fue mi compañero de proyecto, con la mejor onda y disponibilidad. A Joaco, por su amistad y su ayuda en momentos difíciles."

Abstract

InterfLan es un lenguaje de especificación de interfaces para entrada de datos en la forma de formularios. Es una tarea difícil la de traducir una especificación descrita en forma narrada a código que cumpla con todas las reglas especificadas. Tanto es así que existen, en otras áreas de la programación, lenguajes intermedios para poder realizar una especificación de alto nivel y luego construir la implementación en base a dicha especificación. Eso es lo que InterfLan intenta resolver. Este nivel de abstracción para la generación de interfaces es relevante para todos aquellos que realizan formularios en base a especificaciones narradas.

Este proyecto se encuentra basado en un proyecto anterior del autor donde se establecen las primeras características del lenguaje y una primera implementación incompleta del compilador. La nueva versión de la solución consta de la gramática actualizada con las reglas para poder escribir en el lenguaje, un verificador de sintaxis, tipos y otras propiedades necesarias para comprobar la corrección del formulario, y el generador de código que compila InterfLan en código *JavaScript*. Además este trabajo agrega un análisis más profundo de dependencias y formaliza la sintaxis y la semántica del lenguaje a través de la sintaxis abstracta, una semántica declarativa y otra operacional. En el proceso de desarrollo investigamos los lenguajes que más se utilizan hoy en día para definir formularios y poder identificar qué carencias existían en dicha área. La etapa de desarrollo estuvo guiada principalmente por el proceso de desarrollo de un compilador sugerido en el curso “Lenguajes de programación"[1] de este título. Luego se estudiaron ejemplos realistas de formularios como el formulario de solicitud para una tarjeta de crédito y un formulario para la configuración de marcapasos extraído de la tesis de Sebastián Degrandi [2].

Concluímos que la implementación de nuestro lenguaje y su compilador tienen muchas ventajas en cuanto a la usabilidad con respecto a las soluciones existentes, pero que queda mucho por hacer para poder demostrar la corrección de los formularios. Además se puede usar el resultado de nuestra solución para convertir una especificación en código InterfLan a otros lenguajes o *frameworks*.

Palabras clave

lenguaje de especificación de interfaces; lenguaje específico de dominio; compilación

Índice

1 Introducción

InterfLan se define como un lenguaje de programación que trata de resolver el problema de implementar interfaces de ingreso de datos partiendo únicamente de la especificación de los datos de entrada y las validaciones necesarias para estas entradas. Para ello se brinda al desarrollador herramientas de alto nivel de abstracción para que la implementación de la interfaz no sea tan diferente a la especificación narrada. Cabe destacar que en la industria existen otros lenguajes y *frameworks* que tratan de abordar este problema, pero InterfLan ha logrado una abstracción superior a los demás.

1.1 Motivación

La principal motivación surge del estudio del trabajo de Degrandi [2], donde se analiza el problema de la especificación de formularios complejos para sistemas como el de programación de marcapasos cardíacos. Allí nos dispusimos a obtener una solución más adecuada a los requerimientos que hoy en día los programadores exigen de los lenguajes de especificación. Es interesante investigar la forma de generar una nueva manera de expresar las restricciones que un formulario puede presentar de forma sencilla y evitando ambigüedades. Sentimos que el trabajo anterior de Degrandi dejó la posibilidad de generalizar la idea principal, o sea brindar al desarrollador un lenguaje lo suficientemente general para poder generar, en base a una especificación, tanto formularios simples como aquellos con validaciones complejas.

También se tomó motivación de varios *frameworks* de *JavaScript*. Entre tantas herramientas analizadas, se tomó motivación de *ReactJS*, *Redux* y *Redux Form*. La primera herramienta brinda al desarrollador la posibilidad de construir interfaces de usuarios desde el punto de vista de los componentes, mientras que las otras dos se centran en el comportamiento de la aplicación a partir del estado de la misma. Particularmente, *Redux Form* es una extensión de *Redux* para manejar de forma óptima el estado de un formulario. Estas herramientas son muy interesantes en la temática de InterfLan ya que comparten atributos con nuestro lenguaje: se basan en los componentes del

formulario y manejan las validaciones de cada uno de ellos de forma independiente y transparente. Ahora la gran diferencia que tiene InterfLan con estos *frameworks* es que igualmente nuestro lenguaje provee mayor nivel de abstracción y una curva de aprendizaje inferior a la que requieren estas herramientas.

En síntesis, la tarea de transformar los requerimientos del cliente o de los interesados desde su versión narrada a código en algún lenguaje de programación no suele ser sencillo y es común que en este pasaje se pierdan algunos detalles. Además de esto último, se suma el hecho que para varios lenguajes de programación que intentan resolver este problema, las habilidades de los desarrolladores deben ser muy buenas para librar con éxito los inconvenientes de esta traducción. Entonces, InterfLan es un lenguaje que propone facilitar la implementación de formularios basados en su especificación ahorrando tiempo en desarrollar las habilidades para dominar el lenguaje sin sacrificar detalles de especificación.

1.2 ¿Qué existe?

Por un lado tenemos los lenguajes de carácter general que permiten generar interfaces gráficas, tanto de escritorio como aplicaciones web. De éstos nos quisimos concentrar en los web, ya que son los más populares hoy en día, dado que la gran mayoría del desarrollo se mueve en esa dirección. Esto no se debe a alguna característica en particular de esa clase de interfaces, sino al hecho de que son simples de integrar en arquitecturas web favoreciendo atributos de calidad tales como disponibilidad, accesibilidad y portabilidad. Pero no queremos quedarnos con los lenguajes de carácter general porque en ellos suele ser necesario tener que declarar estructuras complejas o propias de la práctica de la programación y mucho código procedural para poder darle el comportamiento deseado a un formulario. En cambio queremos quedarnos con los lenguajes y *frameworks* que nos permiten de manera natural, y mayormente declarativa, describir el comportamiento que el formulario tiene cuando recibe estímulos. Los que hemos encontrado relevantes y que cumplen con estas características son *UIML* (*User Interface Markup Languaje*), *AngularJS*, *KnockoutJS* y *ReactJS*. Todos permiten especificar la mayoría de las restricciones o el comportamiento que un formulario podría llegar a tener de manera declarativa y natural. También consideramos estudiar *RoR* (*Ruby On Rails*) y *Django*, ya que parecían prometedores, pero descubrimos que los formularios en esos *frameworks* naturalmente están atados a modelos de datos con las validaciones en ellos, y por lo tanto no eran declarativos. Uno para poder utilizarlos debería especificar, en general, uno

o varios objetos que representan la entrada de datos o caer en especificar el comportamiento del formulario de la misma manera que en un lenguaje de carácter general. También debemos hablar de la tesis de Degrandi, que propone una solución al mismo problema que estamos atacando pero pone en prioridad aspectos distintos que nosotros. Es cierto que su solución es declarativa, pero no tiene una forma natural de expresar las restricciones. De todos modos una comparación más extensa se presentará en la siguiente sección.

1.3 Comparación y Contribución

Para empezar a comparar las opciones que ya existen, necesitamos hablar de qué características son deseables en un lenguaje de este estilo, o por lo menos de aquellas que nosotros consideramos son las más valiosas. El lenguaje que necesitamos debe ser:

- Fácil de entender.
- Sencillo de escribir.
- Declarativo.
- Con suficiente potencial y flexibilidad como para poder expresar cualquier tipo de relación que tenga sentido entre campos.
- Se debe poder comprobar que las relaciones entre los campos tienen sentido (por lo menos en su gran mayoría).

Partiendo de esto podemos analizar qué ventajas y desventajas tienen los lenguajes y *frameworks* antes mencionados.

1.3.1 UIML

```
...
<behavior>
   <rule>
     <condition>
       ...
     </condition>
     <action>
       ...
     </action>
```

```
  </rule>
</behavior>
...
```

Este lenguaje [3] permite la especificación de interfaz de usuario mediante etiquetas. Pero a diferencia de HTML, no es un formulario con controles estáticos, sino que se puede definir comportamiento frente a estímulos dentro de la etiqueta `<behavior></behavior>`. El desarrollador define en una parte de la estructura los campos, los atributos que tienen, y luego el comportamiento que van a tener mediante reglas, cada una de las cuales es dada por una condición y una acción. Las acciones suelen cambiar alguna propiedad de uno o más campos, permitiendo así cambiar su valor, su visibilidad, etc. Al parecer todos los tipos de restricciones se pueden expresar en él, además es declarativo y es muy parecido a HTML, por lo que aprender a escribir en él no es una tarea compleja. Sin embargo tiene algunos problemas. Es complejo entender las relaciones que hay entre los campos ya que todas van en la sección de comportamiento y no junto a los campos en sí. Se necesita conocer muchas etiquetas para poder completar la estructura y además ésta es muy larga. En el fragmento de código de arriba se muestran solamente las etiquetas que rodean la sección de comportamiento, pero para especificar solamente la condición es necesario conocer al menos cinco etiquetas más que estarán anidadas dentro de la condición, sin contar las de la acción. Otro problema que presenta es que debido a que todo el comportamiento va en la misma sección, pero no con un orden en particular, puede ser que se redefina comportamiento o que hayan conflictos que no son simples de visualizar por la separación de los campos y el comportamiento. También está el hecho de que es necesario usar demasiados símbolos para poder declarar pocas cosas debido a la estructura de las etiquetas.

1.3.2 *AngularJS*

```
...
<table>
  <tr><th>Row number</th></tr>
  <tr ng-repeat="i in [0, 1, 2, 3, 4, 5, 6, 7]"><td>{{i}}</td></tr>
</table>
...
```

Este es un *framework* para *JavaScript* y HTML, que permite agregar controladores con características particulares como el *two-way-binding* y etiquetas con comportamiento en HTML. La idea de *AngularJS* [4] es mantener un modelo que esté en un controlador, asociado a una vista. Los valores de

la vista se ven afectados por los cambios en el modelo y viceversa (característica conocida como *two-way-binding*). Esto hace que definir comportamiento condicional en *AngularJS* sea muy simple y que los cambios en el formulario se visualicen al instante. Debido a que los valores están en sincronía en el modelo y en la vista, es muy simple manejar relaciones del lado del modelo. Entonces se puede realizar habilitaciones y calcular campos en base a otros valores. Además de eso, la librería básica de *AngularJS* provee muchas validaciones comunes a todos los formularios, por ejemplo de la presencia de valores en un campo, validaciones con expresiones regulares (*regex*), etc. Pero tiene algunas desventajas. Por ejemplo, las estructuras o modelos que se declaran en el controlador no suelen ser las más intuitivas y uno necesita de un poco de experiencia para darse cuenta de cuál es su funcionalidad o para idearlas. Otro inconveniente es que la estructura se compone de HTML con anotaciones y de uno o más archivos de *JavaScript*, pero las validaciones no se concentran en uno solo, sino que algunas de ellas van como anotaciones en el HTML y otras van implícitas en el código *JavaScript*. Y a pesar de tener la gran mayoría de las relaciones posibles entre valores, no encontramos una manera nativa del *framework* para poder expresar una relación bidireccional entre dos campos.

1.3.3 *KnockoutJS*

```
...
var ViewModel = function(first, last) {
    this.firstName = ko.observable(first);
    this.lastName = ko.observable(last);

    this.fullName = ko.pureComputed(function() {
        return this.firstName() + " " + this.lastName();
    }, this);
};
...
```

KnockoutJS [5], al igual que *AngularJS*, es un *framework* de *JavaScript* y HTML. No sólo eso, sino que tiene muchas similitudes en cuanto al manejo de los valores. Presenta también *data bindings*, como el *two-way-binding* de *AngularJS*, pero en *KnockoutJS* uno debe especificar qué valores tienen el *binding* y qué tipo de relación tienen. Por ejemplo, un valor puede tener *binding* con un campo y que su valor dependa del definido en el controlador y en la vista o puede ser que sea un valor computado en base al valor de otros campos (caso en el cual sería un computado). Esto tiene sus ventajas y desventajas, ya que el programador tiene que incorporar más palabras

para poder definir lo mismo que en el otro *framework* y es necesario entender qué significado tiene cada tipo de relación, pero tiene más claro cómo es la relación de ese valor con quienes lo observan. La otra ventaja que tiene *KnockoutJS* es la variedad de tipos de validaciones o *bindings* que pueden tener los campos en el HTML. Con todos los *bindings* que tiene *KnockoutJS* podemos expresar todos los tipos de relaciones entre campos que queríamos expresar. Pero al igual que *AngularJS* tiene el problema de que a veces es necesario hacer estructuras como modelos que no tienen una extrapolación directa con algún concepto del formulario, sino que son artilugios para poder generar el comportamiento deseado. A pesar de eso es uno de los que tienen menor curva de aprendizaje y es bastante intuitivo, pero no llega al nivel que deseamos. Una de las diferencias importantes que tiene *KnockoutJS* con respecto a Angular es que en *AngularJS* los cambios de *binding* se generan con cada evento disparado sobre un campo, pero en *KnockoutJS* sólo se disparan las validaciones y las relaciones cuando uno pierde el foco sobre un campo.

1.3.4 *ReactJS* y *Redux Form*

```
...
class MyForm extends Component {
    render () {
        const { handleSubmit } = this.props

        return (
            <form onSubmit={handleSubmit}>
                <Field
                  name='firstName'
                  component='input'
                  type='text'
                  placeholder='First Name'
                />
            </form>
        )
    }
}
...
```

ReactJS [6] es un *framework* de *JavaScript* al igual que los anteriores, pero que se focaliza en construcción de la interfaz de usuario a través de componentes. Los componentes son unidades lógicas que encapsulan al componente de interfaz (lo que el usuario realmente ve) en conjunto con su comportamiento. De esta forma *ReactJS* logra optimizar el proceso de actuali-

zación o de creación de un componente, modificando solamente lo que sea necesario mediante el uso de estados. *Redux* [7] y *Redux Form* [8] son dos herramientas que suelen utilizarse frecuentemente junto a *ReactJS*, de tal forma que: la primera tiene como objetivo el manejo de estados a nivel de la aplicación y provee mecanismos para ejecutar transiciones a muy alto nivel para el desarrollador, ya que todo el manejo de las mismas se hace de forma transparente y automática; y el segundo agrega el manejo de los estados de un formulario y sus campos de forma automática utilizando *Redux* para el manejo del mismo. En principio, todas estas herramientas parecen ser muy convenientes para InterfLan, ya que podríamos decir que comparten la mayoría de los atributos deseados. Todos ellos trabajan sobre *ReactJS* por lo que se vuelve un *framework* declarativo, ya que el manejo de los campos de un formulario está abstraído en componentes. Además, estas herramientas para construir un formulario son muy flexibles ya que proporcionan solamente el manejo de los estados del formulario sin involucrar la interfaz gráfica ni a comportamientos personalizados que resultan ser relativamente fácil de introducir. Igualmente, a pesar de tener muy buenos atributos para el problema planteado, la curva de aprendizaje de tres herramientas para poder "especificar" formularios es muy alta ya que hay que dominarlas bien para que la flexibilidad que brindan sea aprovechable al cien por ciento.

1.3.5 *Ruby On Rails - Django*

```
... (Ruby On Rails)
class Article < ApplicationRecord
  validates :title, presence: true,
                    length: { minimum: 5 }
end
...
... (Django)
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
...
```

Finalmente están los *frameworks RoR* [9] (*Ruby On Rails*) y *Django* [10] que se forman sobre *Ruby* y *Python* respectivamente, pero que comparten muchas características. Estos son *frameworks* diseñados para generar soluciones de manera muy sencilla y rápida. Construir aplicaciones web complejas es muy simple en estos lenguajes, pero por ello mismo no se concentran en

la parte gráfica en cuanto al ingreso de datos, sino más bien en los modelos que representan las entidades de la aplicación. A pesar de que algunas de las validaciones que creemos necesarias parecen poder expresarse de forma sencilla, esto no es tan así porque la parte gráfica en estos lenguajes se genera en el servidor de manera estática y luego se envía al navegador la vista del formulario. En ésta se suelen utilizan otros lenguajes o *frameworks* para hacer dinámica la interacción con el usuario, por lo que no podemos generar cambios en el formulario a medida se van ingresando datos. Sin embargo, queremos destacar que la sintaxis de ambos lenguajes es muy buena. Es simple de entender, clara, amigable para el desarrollador y para el cliente (ya que se puede leer de la misma forma que un texto en prosa) y utiliza pocos símbolos.

1.3.6 Lenguaje de especificación para marcapasos

El lenguaje propuesto por Degrandi[2] en su tesis tiene características muy positivas. Por un lado es un lenguaje con un poder expresivo muy grande debido a que todos los campos tienen un dominio y ése es el cual se puede calcular en base a otros, permitiendo así generar todas las validaciones posibles. También es interesante que todos los valores en los campos aparezcan como "valores seleccionables", ya que impide que el usuario ingrese cualquier valor en el input. Sin embargo tiene características que no nos parecieron positivas.

El código necesitaba de mucho conocimiento de *C* para poder generar los dominios y hasta a veces se debían generar de manera procedural en lugar de declarativa. Otro aspecto negativo es que los controles son solamente declarados en un sector y luego en otra parte se declaran las restricciones, lo cual no establece un vínculo claro entre el campo y todas sus validaciones o restricciones. Además las estructuras necesarias para poder acceder a los valores de otros campos o para declarar algunas expresiones son muy largas y aumentan el tamaño del código de manera innecesaria.

Además observamos que todas las decisiones de diseño que fueron tomadas en su lenguaje apuntaban únicamente a la especificación del formulario de los marcapasos y no pensó en un público más amplio.

En conclusión no encontramos un lenguaje o herramienta que se adecue completamente a los atributos que queremos favorecer. Igualmente la herramienta que más poder de expresividad nos da para el problema a resolver es *Redux Form (ReactJS)* aunque sería conveniente tomar la simpleza en la sintaxis de otros lenguajes como *Ruby* o *Python* que ayudaría a disminuir

la curva de aprendizaje de las herramientas que mejor se acercan a nuestro problema.

1.4 Antecedente

El proyecto de InterfLan [11] nació a medidados de 2016 bajo el marco del trabajo integrador para la obtención del título de Licenciado en Ingeniería de Software, conjuntamente con Joaquín Silveira. Debido a su corta duración, el trabajo se centró en un primer avance sobre el estado del arte de este tipo de lenguajes, la búsqueda de atributos de calidad deseados para nuestro lenguaje y una primera implementación de la sintaxis. En ese trabajo se dejaron claras varias decisiones que continúan en esta versión. A partir de la investigación del estado del arte sobre lenguajes o herramientas que permitieran expresar lo que InterfLan pretende, se concluyó que InterfLan debía converger a compilar a una aplicación web. Esto se debe a que hoy en día la industria se mueve en torno a aplicaciones distribuídas y con base web dejando de lado las aplicaciones de escritorio, y que además la comunidad de desarrolladores es tan grande que permite encontrar varias herramientas y otros proyectos que brindan a la misma comunidad el poder solucionar cualquier problema.

En base a las características planteadas para el lenguaje, el trabajo anterior concluyó con una implementación de la gramática del mismo, que hasta el momento mantiene los mismos lineamientos, y se establecieron las reglas de tipos y validaciones de estructuras que denotan aquellos programas válidos. Aún así, se dejó constancia de rever la verificación de algunas dependencias que el lenguaje no dejó cien por ciento aseguradas, y hacer así más robusto al lenguaje.

Por último, es importante destacar que el presente documento tiene como objetivo aportar la descripción del lenguaje en su totalidad para que el lector pueda tener todo el contenido relativo al lenguaje en un único documento.

1.5 Objetivos

Dado el antecedente de InterfLan mencionado anteriormente, los planteos que quedaron en “trabajo a futuro", se plantearon los siguientes objetivos que a lo largo del documento se desarrollarán.

1. Completar el diseño del InterfLan con una semántica formal explícita que sirva de base para razonar sobre las propiedades del lenguaje.

2. Completar la implementación con un generador de código HTML - *JavaScript*. El objetivo será entonces disponer de una herramienta efectivamente usable en la práctica.

3. Rever el tratamiento de las restricciones a las dependencias de cálculo entre campos para lograr un resultado más completo que en la versión precedente.

2 Guía del Lenguaje

Para poder analizar la estructura del lenguaje y para que el lector comprenda el uso del mismo decidimos documentar el proceso de especificación de los siguientes formularios, para tratar de abarcar todo lo que se puede expresar en él.

2.1 Ejemplo: Solicitud de Línea de crédito

Este formulario (**Figura 2.1**), como sus campos delatan, es para una solicitud de una línea de crédito personal. Consta de algunos campos sencillos, como el nombre y dirección, y de otros que no lo son tanto, como la fecha de nacimiento, la fecha de casamiento, la ciudad y el país de nacimiento y el ingreso total para la solicitud del monto de la línea de crédito. Es importante que expliquemos los campos complejos antes de empezar con la especificación, ya que ése sería el proceso natural por el cual la especificación pasaría. Una redacción de las restricciones que debe poseer el formulario anterior podría ser la siguiente:

- Ninguno de los campos del formulario puede ser vacío, a excepción de la fecha de matrimonio o concubinato; de no ser alguno de los recién mencionados la situación actual del solicitante.
- El correo electrónico debe ser un correo electrónico sintácticamente bien formado.
- La cédula de identidad debe pasar por la función que verifica el último dígito de la misma.
- La fecha de nacimiento debe corresponder con la de una persona mayor de edad (o sea mayor de 18 años).
- La edad no se podrá ingresar manualmente, sino que será calculada en el momento en el que se ingrese la fecha de nacimiento.

Figura 2.1: Formulario de ejemplo.

- En caso de que el estado civil del solicitante sea“Concubino" o “Casado", se deberá habilitar el campo de la fecha de matrimonio o concubinato; de otra forma el campo estará deshabilitado.
- Una vez que se seleccione el país de nacimiento, se mostrarán las ciudades correspondientes a dicho país en la opción de “Seleccionar Ciudad".
- Tanto si se ingresa el límite de crédito o los ingresos nominales, será calculado el otro valor. En caso de ingresarse los ingresos nominales, el máximo de la línea de crédito será el doble de dicho valor. En cambio si se ingresa el valor máximo de la línea de crédito deseada, se calculará los ingresos nominales mínimos necesarios para obtener el crédito, como la mitad de la línea de crédito.

Ahora, teniendo la especificación en español, podemos pasar a crear el formulario. Debemos definir la estructura que contiene los campos para la entrada de datos. Podríamos ponerle como nombre `linea_de_credito`.

```
form linea_de_credito let

end
```

Código 2.1: Definición del formulario.

Además, si queremos que el formulario muestre un título específico una vez compilado podemos agregar un texto en la definición del nombre del formulario.

```
form "Solicitud de Línea de Crédito" as linea_de_credito let

end
```

Código 2.2: Título a mostrar del formulario.

Sin embargo, un formulario sin campos no tiene sentido, así que agreguemos un campo del formulario. La declaración de los campos es similar a la del formulario, sólo que se deben especificar algunos datos más. Por ejemplo, lo siguiente constituiría una declaración de un campo para ingresar el nombre del solicitante.

```
input "Nombre" as nombre:string let

end
```

Código 2.3: Definición de un control.

La definición de un control está compuesta por:

- **Tipo de Campo:** Aquí se seleccionará la "forma" del campo según su interacción con el usuario. Se puede seleccionar entre: `input` para definir un campo de ingreso de datos, `label` para definir un campo que sólo muestra determinado dato sin que el usuario pueda editarlo, y `select` para definir un campo donde el usuario podrá seleccionar únicamente un valor del dominio.

- **Nombre:** Se definirá el nombre del campo para constituirlo como variable dentro del formulario y opcionalmente se podrá especificar un nombre distinto para la vista al usuario. Esto último se lo conoce como *caption* (`"Texto a mostrar" as variable`) cuyo uso se puede observar en el **Código 2.2**.

- **Tipo de dato:** Todo campo deberá tener acompañado de su nombre el tipo de datos que aceptará. Este puede ser uno de los primitivos (`integer`, `bool`, etc.) o un enumerado (que veremos más adelante).

- **Características y restricciones:** Por último tendremos que especificarán qué restricciones, validaciones y dependencias tendrán, las cuales iremos agregando a continuación.

Continuando con la definición de todos los campos, tendríamos lo siguiente:

```
form linea_de_credito let
    input "Nombre" as nombre:string let
    end

    input "Apellido" as apellido:string let
    end

    input "Dirección de correo  electr ónico" as email:string let
    end

    input "Cédula de identidad" as cedula:integer let
    end

    input "Fecha de Nacimiento" as f_nacimiento:datetime let
    end

    label "Edad" as edad:integer let
    end

    select "Estado civil" as ecivil:<?> let
    end

    input "Fecha concubinato o matrimonio" as f_mat:datetime let
    end

    select "País de nacimiento" as p_nacimiento:<?> let
    end

    select "Ciudad de nacimiento" as c_nacimiento:<?> let
    end

    input "Mínimo ingreso nominal" as ingresos:integer let
    end

    input "Máximo línea de crédito" as credito:integer let
    end
end
```

Código 2.4: Definición de todos los controles.

Nótese que la edad no se declaró como un `input` sino como un `label` para cumplir con la especificación de que no debe ser un campo que se pueda

modificar manualmente. También es importante ver que no hemos podido declarar el tipo de los controles `select` ya que su tipo no se corresponde con ninguno de los primitivos. Por lo cual debemos declarar los tipos para los `select`. Para el estado civil y los países simplemente necesitamos un enumerado con los valores válidos de la siguiente manera:

```
enum estado_civil [("Casado/a",CAS),("Soltero/a",SOL),
                   ("Concubino/a",CON),("Viudo/a",VIU),
                   ("Divorciado/a", DIV)] ;

enum paises [("Uruguay",UY), ("Brasil", BR), ("Argentina", AR)] ;
```

Código 2.5: Definición de enumerados.

Los "valores" de los enumerados están compuestos por un par ordenado (*caption*, valor), para poder diferenciar la referencia interna en el código y lo que se muestra al usuario.

Para las ciudades podríamos hacer lo mismo, sólo que tendríamos el inconveniente de que no importa que país se elija, aparecerán todas las ciudades. Así que debemos crear un enumerado que dependa de el país que se ha seleccionado.

```
enum ciudades let
    domain (case p_nacimiento of
                paises.UY -> [("Montevideo", MVD), ("Rocha", ROC),
                              ("Canelones", CAN)] ;
                paises.BR -> [("Brasilia", BRS),("Fortaleza", FRT),
                              ("Recife",RCF)] ;
                paises.AR -> [("Buenos Aires", BAS), ("Santa Fe",STF),
                              ("Córdoba", CBA)]
            end) ;
end
```

Código 2.6: Definición de enumerado con dependencia

Analizando la declaración anterior, podemos observar la primera sentencia dentro de un bloque en nuestro lenguaje. Esa sentencia define que el dominio del tipo enumerado `ciudades` se define por casos en base al valor del control `p_nacimiento`. Luego se define qué es lo que pasa en cada uno de los casos posibles del `select p_nacimiento`. Cada una de las opciones se llama **rama** y dan un nuevo enumerado para cada caso posible, definiendo así los valores que va a tomar el enumerado en cada instante. Repasando un poco podemos llegar a que la sección de los `select` se define así:

```
    ...
```

```
select "Estado civil" as ecivil:estado_civil let
end
...
select "País de nacimiento" as p_nacimiento:paises let
end

select "Ciudad de nacimiento" as c_nacimiento:ciudades let
end

enum estado_civil [("Casado/a",CAS),("Soltero/a",SOL),
              ("Concubino/a",CON),("Viudo/a",VIU),
              ("Divorciado/a", DIV)] ;

enum paises [("Uruguay",UY), ("Brasil", BR), ("Argentina", AR)] ;

enum ciudades let
    domain (case p_nacimiento of
                paises.UY -> [("Montevideo", MVD), ("Rocha", ROC),
                     ("Canelones", CAN)] ;
                paises.BR -> [("Brasilia", BRS),("Fortaleza", FRT), (
                    "Recife",RCF)] ;
                paises.AR -> [("Buenos Aires", BAS), ("Santa Fe",STF
                    ), ("Córdoba", CBA)]
            end) ;
end
...
```

Código 2.7: Definición de los select y enums

Lo que resta es generar las validaciones para los demás controles. Para poder controlar que ninguno de los campos se encuentre vacío luego de ingresar los datos, debemos agregar la siguiente sentencia en todos los campos:

```
input "Nombre" as nombre:string let
    not_empty;
end
```

Código 2.8: Utilización de sentencia not_empty.

Para cumplir con la restricción de la validación del correo electrónico, introducimos la sentencia `regex`, de la siguiente manera:

```
input "Dirección de correo  electr ónico" as email string let
    not_empty ;
    regex "(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0
        -9-.]+$)" ;
```

```
end
```

Código 2.9: Utilización de la sentencia regex.

Para validar la fecha de nacimiento podemos hacer uso de un rango que deje claro que la fecha es válida y el solicitante es mayor de 18.

```
input "Fecha de Nacimiento" as f_necimiento:datetime let
    not_empty ;
    range (subs_years(now, 150))..(subs_years(now, 18)) ;
end
```

Código 2.10: Utilizacón de la sentencia range.

De esta manera nos aseguramos que la persona no sea mayor que 150 años ni menor de 18. Las operaciones de fechas se realizan de la siguiente manera: `subs_` o `add_`, la unidad sobre la que se va a operar (*years*, *months*, *days*), la fecha base de la operación y la cantidad que se va a aumentar o disminuir.

También, podemos simplificar esta validación si en vez de validar la fecha, validamos el campo `label` que contiene la edad. En este caso, la validación es más sencilla ya que solamente debemos verificar que ese número sea mayor o igual a 18.

```
label "Edad" as edad:integer let
    check (self >= 18) ;
end
```

Código 2.11: Verificación de la edad.

Continuando con las restricciones, debemos introducir las funciones para poder otorgarle a quien realice la especificación el potencial de especificar cosas que se encuentran por fuera de nuestro lenguaje. Permitimos realizar funciones con cuerpo escrito en *JavaScript*, que es un lenguaje popular y flexible. La estructura de las funciones es la siguiente:

```
function verificar_cedula:bool (cedula:integer) let
    "cuerpo de la función en JavaScript"
end
```

Código 2.12: Definición de una función.

El nombre de la función está acompañado, luego de los ":", por el tipo que retorna la función, luego continúa la lista de parámetros que recibe la función con sus correspondientes tipos y finalmente el cuerpo que contiene

el código *JavaScript* entre comillas. Ahora podemos realizar las validaciones del cálculo de la edad y la verificación de la cédula.

```
function verificar_cedula:bool (cedula:integer) let
    "cuerpo de la función en JavaScript"
end
input "Cédula de identidad" as cedula:integer let
    not_empty ;
    check (verificar_cedula(self)) ;
end
...
function calcular_edad:integer (fnac:datetime) let
    "cuerpo de la función en JavaScript"
end
label "Edad" as edad:integer let
    calculate (calcular_edad(f_nacimiento)) ;
end
```

Código 2.13: Definición de funciones de cédula y edad.

Veamos qué cosas se han agregado a la sintaxis. Además de las funciones que verifican la cédula y que calculan la edad del solicitante, necesitamos declarar la validación de la cédula en el cuerpo haciendo uso de la sentencia `check` que contiene una expresión booleana. También debemos declarar que el valor del `label edad` se calcula mediante la expresión que le sigue. Es importante destacar también la expresión *self*, la cual hace referencia al valor del campo en el cual la expresión se encuentra. Continuando con las validaciones para la fecha de matrimonio o concubinato, debemos declarar que se debe habilitar solamante cuando el estado civil del solicitante sea concubino o casado. Y sería conveniente también que esta fecha sea coherente con la fecha de nacimiento, y que el campo sea obligatorio cuando esté habilitado.

```
input "Fecha concubinato o matrimonio" as f_mat:datetime let
    not_empty (ecivil in [CAS,CON]) ;
    enable (ecivil in [CAS,CON]) ;
    range (add_years f_nacimiento 14)..(now) ;
end
```

Código 2.14: Utilización de la sentencia enable.

Finalmente lo que resta es la relación entre la línea de crédito y los ingresos. Debido a que la relación es bidireccional, se debe establecer una manera de calcular el ingreso mínimo en base a la línea de crédito y otra forma de calcular la línea de crédito en base al ingreso. Para esto, se utilizan las dependencias `calculate` que hemos visto anteriormente, pero agrupando las

dos definiciones de los campos en un bloque `equity`. Este bloque define que los campos definidos a continuación se encuentran en una relación bidireccional, e indica al lenguaje que la circularidad introducida a nivel del cálculo es deseada.

```
equity let
    input "Mínimo ingreso nominal" as ingresos:integer let
        not_empty ;
        calculate (credito / 2) ;
    end

    input "Máximo línea de crédito" as credito:integer let
        not_empty ;
        calculate (ingresos * 2) ;
    end
end
```

Código 2.15: Utilización de equity.

Se deja en el **Anexo B.1** el formulario completo al finalizar este ejemplo.

3 Descripción del Lenguaje

En este capítulo nos centraremos en describir de manera más formal cómo se escribe en el lenguaje, qué estructuras se manejan y cómo se comporta un formulario con las acciones del usuario.

3.1 Sintaxis

3.1.1 Sintaxis abstracta

Explicaremos la estructura sintáctica del lenguaje, como por ejemplo de qué manera se agrupan los distintos componentes de un formulario de InterfLan. Por más detalles en la sintaxis, el lector puede dirigirse al **Anexo A** donde se encuentra la sintaxis concreta del lenguaje.

Primero, es importante resaltar los tipos de datos que están disponibles en el lenguaje ya que se hará mención a ellos más adelante. En InterfLan el tipo de datos puede referir a un tipo de datos primitivo (`bool`, `integer`, `float`, `datetime` y `string`), o a un tipo de datos enumerado definido en la estructura de un formulario.

Un formulario está compuesto por el nombre del mismo y una lista de definiciones. El nombre tiene un identificador y además se le puede vincular un *string* como título.

Las definiciones son los elementos que conforman el cuerpo del formulario en sí. Ellas introducen cuatro clases de componentes distintas:

- **Campos:** Son las únicas definiciones que el usuario puede visualizar. Su función es recibir el ingreso de datos por parte del usuario. Los campos están compuestos por el tipo de campo (este puede ser `input`, `label` o `select`), el nombre (definido de la misma forma que para el formulario), el tipo de datos que acepta el campo y una lista de sentencias.

- **Funciones:** Estas definiciones sirven al desarrollador para realizar operaciones y validaciones que no se pueden expresar con el lenguaje en sí, como la definición de procedimientos o rutinas para la validación de un campo. Están compuestas por un identificador, el tipo de datos de retorno, los tipos e identificadores de los argumentos y el cuerpo de la función expresada en *JavaScript*. Cabe destacar que no es posible implementar procedimientos recursivos ni el uso del valor de los campos en el cuerpo de la función.

- **Enumerados:** Introducen un nuevo tipo de datos finito al formulario. Estos enumerados pueden ser independientes o dependientes del valor de un campo. Los enumerados independientes se componen por un identificador (el nombre del nuevo tipo) y una lista de los valores, donde cada valor es un par *string* - identificador (el primero define el nombre del valor hacia el usuario y el segundo el valor que lo identifica en el código). Por otro lado, los enumerados dependientes, como su nombre lo indica, los valores permitidos por el tipo cambian según el valor del otro campo. Se compone también por un identificador y una expresión `case` para determinar las listas de valores posibles conforme al valor del campo del cual depende.

- **Relaciones de equidad:** Esta definición establece una dependencia de cálculo circular entre dos campos de tipo `input`. La definición de esta relación contiene las definiciones de los dos campos a relacionar.

Las sentencias de los campos denotan características de los mismos como validaciones de datos, habilitación de un campo o el cálculo de un valor a partir de otro campo.

- **Validación de datos:** Estas sentencias verifican que la entrada de datos al campo en donde está contenida sea correcta. Para ello existen cuatro tipos de validaciones:

  - `not_empty`: Esta sentencia valida la obligatoriedad de un campo, cuando es requerido que el campo contenga un valor. Para ello la sentencia está compuesta por una expresión que evalúa si el campo es obligatorio o no.
  - `regex`: La sentencia `regex` evalúa que el valor del campo de tipo *string* cumpla con una expresión regular. Entonces esta sentencia se compone por un *string* que contiene la expresión regular a evaluar y opcionalmente otro *string* con el mensaje de error a mostrar al usuario.

  - `range`: Aquí se evalúa que el valor de un campo se encuentre en un intervalo de valores definidos en esta sentencia. Por lo cual `range` está definida con dos expresiones que denotan las cotas del intervalo y opcionalmente acompañadas de un *string* con el mensaje de error a mostrar al usuario.
  - `check`: Esta sentencia es la más genérica de todas las validaciones: sólo es necesario brindar una expresión booleana a evaluar, siendo suficiente que esa expresión evalúe a `True` para que la validación sea correcta. Además, esta sentencia puede ir acompañada de un *string* con el mensaje de error a mostrar al usuario.

- **Habilitación de un campo**: Esta sentencia determina cuándo un campo está habilitado al usuario para recibir entrada de datos, por lo que está compuesta por una expresión booleana que determina si el campo pasa a estar habilitado o no.

- **Cálculo de valor:** Estas sentencias se utilizan para determinar el valor de un campo de tipo `label` o un campo de tipo `input` en relación de equidad (`equity`) a partir de una expresión. Muchas veces el cálculo puede implicar que éste se realice en base al valor de otro campo. Entonces, esta sentencia está compuesta por una expresión del mismo tipo que el campo, la cual determina el valor del mismo.

Las expresiones son los componentes más elementales del lenguaje, son los que forman los valores de InterfLan. Para describirlas simplemente basta agruparlas por su nivel de evaluación:

- **Literales:** Aquí se encuentran los valores más elementales que no necesitan ninguna evaluación, son directamente valores de un tipo de datos o un alias. Por ejemplo una fecha (`datetime`) es una expresión literal, y el alias `now` es un literal que indica la fecha de este momento.

- **Variables:** En este caso las variables están constituidas por la mención al identificar un campo del formulario que denota el valor actual del mismo.

- **Operaciones aritméticas:** Están comprendidas las operaciones habituales para los tipos de numéricos, además de estar sobrecargada la suma para los *strings* y de esa manera expresar la concatenación de los mismos.

- **Operaciones booleanas:** Están comprendidas las operaciones de este tipo con los operadores lógicos (*and*, *or*, *not*) y los operadores de comparación (<, <=, ==, etc.).

- **Aplicación de funciones:** Aquí es donde se ejecutan las funciones definidas anteriormente, es el momento donde los argumentos de la función toman expresiones y se genera un resultado.

- **Estructura de control `case`:** Esta expresión es el equivalente a la estructura `switch` en otros lenguajes y tiene como objetivo principal la definición de enumerados dependientes, aunque también pueden ser utilizados en cualquier otro lugar donde sea necesario decidir según el valor de una expresión.

3.1.2 Reglas sintácticas y de verificación de tipos

Las relgas tienen el siguiente formato, donde $P_1, ..., P_n$ son las condiciones que se tienen que cumplir o premisas, C es la conclusión o la culminación exitosa de la validación y $S_1, ..., S_m$ las condiciónes adicionales que se tienen que cumplir.

$$\frac{P_1, ..., P_n}{\mathrm{C}} \, S_1, ..., S_m$$

Para poder realizar las validaciones de los tipos en las reglas precedentes es necesario generar un contexto que en las mismas es nombrado Γ. Este contexto está conformado por cuatro grandes secciones.

- Una lista de los campos disponibles en el formulario junto con su tipo para poder verificar las variables dentro de las expresiones, las cuales llevan el nombre de los controles.

- Una lista de las funciones con sus respectivas firmas (tipo de devolución de la función y lista ordenada de los tipos de los parámetos).

- Una lista de los enumerados junto con los valores que lo componen.

- Finalmente un grupo de grafos con las dependencias generadas entre los distintos campos, un grafo por cada tipo de sentencia en la que se genera una dependencia, para validar la ausencia de circularidades en las relaciones.

Para hacer uso de los elementos asociados a los nombres de cada estructura (campos, funciones y enumerados), podemos pedirle al contexto que busque por el nombre de la estructura de la forma $\Gamma(\{nombre\})$.

A continuación se especifican las reglas de validación sintáctica y de los tipos en el código:

- **Verificación sintáctica de un formulario:**

$$\frac{\Gamma \vdash [d_1, d_2, ..., d_n] \text{ ; valid}}{\text{form n } [d_1, d_2, ..., d_n] \text{ ; valid}}$$

Figura 3.1: Declaración del formulario

- **Verificación sintática de las definiciones:**

$$\frac{\Gamma, self : t \vdash [s_1, s_2, ..., s_n] \text{ ; valid}}{\Gamma \vdash \text{input n t } [s_1, s_2, ..., s_n] \text{ ; valid}} \text{ n isn't in a Calculate Dep.}$$

Figura 3.2: Definición de un campo input

$$\frac{\Gamma, self : t \vdash [s_1, s_2, ..., s_n] \text{ ; valid}}{\Gamma \vdash \text{label n t } [s_1, s_2, ..., s_n] \text{ ; valid}} \text{ n is in at least one Calculate Dep.}$$

Figura 3.3: Definición de un campo label

$$\frac{\Gamma, self : t \vdash [s_1, s_2, ..., s_n] \text{ ; valid}}{\Gamma \vdash \text{select n t } [s_1, s_2, ..., s_n] \text{ ; valid}} \begin{array}{c}\text{t is enumerate} \\ \text{n isn't in a Calculate Dep.}\end{array}$$

Figura 3.4: Definición de un campo select

$$\frac{}{\Gamma \vdash \text{enum n } [v_1, v_2, ..., v_n] \text{ ; valid}} \begin{array}{c}[v_1, v_2, ..., v_n] \text{ has not} \\ \text{repeated values}\end{array}$$

Figura 3.5: Definición de un enumerado listando sus valores

$$\frac{\Gamma \vdash \text{case e } [e_1 \to v_1, ...e_n \to v_n] : \text{enumerate}}{\Gamma \vdash \text{enum n let domain (case e } [e_1 \to v_1, ...e_n \to v_n]); \text{ end valid}}$$

Figura 3.6: Definición de un enumerado dependiente de otro campo

$$\frac{\Gamma \vdash (\text{input } n_1\ t_1\ ss_1)\ ;\ \text{valid},\ \Gamma \vdash (\text{input } n_2\ t_2\ ss_2)\ ;\ \text{valid}}{\Gamma \vdash \text{equity (input } n_1\ t_1\ ss_1)\ \text{(input } n_2\ t_2\ ss_2\ ;\ \text{valid}}\ n_1,\ n_2 \text{ are in Calculate Dep.}$$

Figura 3.7: Definición de una relación equity entre dos campos

- **Verificación sintáctica de las sentencias:**

$$\frac{}{\Gamma, self : t \vdash \text{not\_empty}\ ;\ \text{valid}}$$

Figura 3.8: Sentencia not_empty, campo obligatorio siempre

$$\frac{\Gamma, self : t \vdash \text{e} :\ \text{bool}}{\Gamma, self : t \vdash \text{not\_empty e}\ ;\ \text{valid}}$$

Figura 3.9: Sentencia not_empty, campo obligatorio según e

$$\frac{}{\Gamma, self : t \vdash \text{regex s}\ ;\ \text{valid}}\ \text{t} = \text{string}$$

Figura 3.10: Sentencia regex, cumplimiento de expresión regular

$$\frac{\Gamma, self : t \vdash \text{e1} :\ t_1,\ \Gamma, self : t \vdash \text{e2} :\ t_2,\ t_1 = t_2}{\Gamma, self : t \vdash \text{range e1 e2}\ ;\ \text{valid}}\ t_1, t_2 \neq \text{enumerate}$$

Figura 3.11: Sentencia range, delimita los valores posibles de un campo

$$\frac{\Gamma, self : t \vdash \text{e} :\ \text{bool}}{\Gamma, self : t \vdash \text{check e}\ ;\ \text{valid}}$$

Figura 3.12: Sentencia check, evaluación de expresión para validar un campo

$$\frac{\Gamma, self : t \vdash \text{e} :\ \text{bool}}{\Gamma, self : t \vdash \text{enable e}\ ;\ \text{valid}}$$

Figura 3.13: Sentencia enable, habilitación de un campo

$$\frac{\Gamma, self : t \vdash \text{e} :\ \text{t'}\ ,\ \text{t'} = \text{t}}{\Gamma, self : t \vdash \text{calculate e}\ ;\ \text{valid}}$$

Figura 3.14: Sentencia calculate, cálculo del valor de un campo

- **Verificación de tipo de las expresiones:**

$$\frac{}{\Gamma, self : t \vdash b : \text{bool}}$$

Figura 3.15: Verificación de tipos de expresiones booleanas

$$\frac{}{\Gamma, self : t \vdash i : \text{integer}}$$

Figura 3.16: Verificación de tipos de expresiones enteras

$$\frac{}{\Gamma, self : t \vdash d : \text{double}}$$

Figura 3.17: Verificación de tipos de expresiones decimales

$$\frac{}{\Gamma, self : t \vdash s : \text{string}}$$

Figura 3.18: Verificación de tipos de expresiones string

$$\frac{}{\Gamma, self : t \vdash \text{now} : \text{datetime}}$$

Figura 3.19: Verificación de tipo del literal now, que indica la fecha y hora actual

$$\frac{}{\Gamma, self : t \vdash \text{"YYYY-MM-DDTHHmm"} : \text{datetime}}$$

Figura 3.20: Verificación de tipos de expresiones fecha y hora

$$\frac{}{\Gamma, self : t \vdash self : \text{t}}$$

Figura 3.21: Verificación de tipo de la expresión que indica el valor actual del campo

$$\frac{\Gamma(x) = t'}{\Gamma, self : t \vdash x : \text{t'}}$$

Figura 3.22: Verificación de tipo de una variable

$$\frac{}{\Gamma, self : t \vdash en.val : \text{ en}} \begin{matrix} en \in \Gamma \\ val \in \Gamma(en) \end{matrix}$$

Figura 3.23: Verificación de tipo de un valor de enumerado

$$\frac{}{\Gamma, self : t \vdash [en_1, ..., en_n] : \text{ enumerate}}$$

Figura 3.24: Verificación de tipo de un enumerado

$$\frac{\Gamma(f) = (t', [t'_1, ..., t'_n]) \;\; \Gamma, self : t \vdash [e_1, ..., e_n] : [t_1, ...t_n]}{\Gamma, self : t \vdash f(e_1, ..., e_n) : \text{ t'}} [t_1, ..., t_n] = [t'_1, ..., t'_n]$$

Figura 3.25: Verificación de tipo de la aplicación de una función

$$\frac{\Gamma, self : t \vdash \text{ e} : \text{ bool}}{\Gamma, self : t \vdash \text{ ! e} : \text{ bool}}$$

Figura 3.26: Verificación de tipo de la operación negación

$$\frac{\Gamma, self : t \vdash e_1 : \text{ t'}, \Gamma, self : t \vdash e_1 : \text{ t'}}{\Gamma, self : t \vdash e_1 * e_2 : \text{ t'}} \text{ t'is integer or double}$$

Figura 3.27: Verificación de tipo de la operación producto entre numerales

$$\frac{\Gamma, self : t \vdash e_1 : \text{ t'}, \Gamma, self : t \vdash e_1 : \text{ t'}}{\Gamma, self : t \vdash e_1 / e_2 : \text{ t'}} \text{ t'is integer or double}$$

Figura 3.28: Verificación de tipo de la operación cociente entre numerales

$$\frac{\Gamma, self : t \vdash e_1 : \text{ t'}, \Gamma, self : t \vdash e_1 : \text{ t'}}{\Gamma, self : t \vdash e_1 - e_2 : \text{ t'}} \text{ t'is integer or double}$$

Figura 3.29: Verificación de tipo de la operación resta entre numerales

$$\frac{\Gamma, self : t \vdash e_1 : \text{ t'}, \Gamma, self : t \vdash e_1 : \text{ t'}}{\Gamma, self : t \vdash e_1 \% e_2 : \text{ t'}} \text{ t'is integer or double}$$

Figura 3.30: Verificación de tipo de la operación módulo entre numerales

$$\frac{\Gamma, self : t \vdash e_1 : t_1, \Gamma, self : t \vdash e_1 : t_2 \ , \max(t_1, t_2)\text{=t'}}{\Gamma, self : t \vdash e_1 + e_2 : \text{t'}} \text{ t'is integer or double or string}$$

Figura 3.31: Verificación de tipo de la operación suma entre numerales y concatenación entre strings

$$\frac{\Gamma, self : t \vdash \text{e} : \text{datetime}, \Gamma, self : t \vdash \text{c} : \text{integer}}{\Gamma, self : t \vdash \text{add\_? e c} : \text{datetime}}$$

Figura 3.32: Verificación de tipo de las operaciones de fecha y hora (adición)

$$\frac{\Gamma, self : t \vdash \text{e} : \text{datetime}, \Gamma, self : t \vdash \text{c} : \text{integer}}{\Gamma, self : t \vdash \text{subs\_? e c} : \text{datetime}}$$

Figura 3.33: Verificación de tipo de las operaciones de fecha y hora (sustracción)

$$\frac{\Gamma, self : t \vdash e_1 : \text{t'}, \Gamma, self : t \vdash e_1 : \text{t'}}{\Gamma, self : t \vdash e_1 < e_2 : \text{bool}} \text{ t'is not enumerate}$$

Figura 3.34: Verificación de tipo del comparador menor

$$\frac{\Gamma, self : t \vdash e_1 : \text{t'}, \Gamma, self : t \vdash e_1 : \text{t'}}{\Gamma, self : t \vdash e_1 > e_2 : \text{bool}} \text{ t'is not enumerate}$$

Figura 3.35: Verificación de tipo del comparador mayor

$$\frac{\Gamma, self : t \vdash e_1 : \text{t'}, \Gamma, self : t \vdash e_1 : \text{t'}}{\Gamma, self : t \vdash e_1 \leq e_2 : \text{bool}} \text{ t' is not enumerate}$$

Figura 3.36: Verificación de tipo del comparador menor o igual

$$\frac{\Gamma, self : t \vdash e_1 : \text{t'}, \Gamma, self : t \vdash e_1 : \text{t'}}{\Gamma, self : t \vdash e_1 \geq e_2 : \text{bool}} \text{ t' is not enumerate}$$

Figura 3.37: Verificación de tipo del comparador mayor o igual

$$\frac{\Gamma, self : t \vdash e_1 : \text{bool}, \Gamma, self : t \vdash e_1 : \text{bool}}{\Gamma, self : t \vdash e_1 \&\& e_2 : \text{bool}}$$

Figura 3.38: Verificación de tipo de la conjunción lógica

$$\frac{\Gamma, self : t \vdash e_1 : \text{boo'}, \Gamma, self : t \vdash e_1 : \text{bool}}{\Gamma, self : t \vdash e_1 || e_2 : \text{bool}}$$

Figura 3.39: verificación de tipo de la disyunción lógica

$$\frac{\Gamma, self : t \vdash \text{e} : \text{t''}, \Gamma, self : t \vdash [e_1, ..., e_n] : [\text{t''},...,\text{t''}], \Gamma, self : t \vdash [v_1, ..., v_n, v'] : [\text{t'},...,\text{t'}]}{\Gamma, self : t \vdash \text{case e } [e_1 \rightarrow v_1, ...e_n \rightarrow v_n] \text{ otherwise} \rightarrow \text{v': t'}}$$

Figura 3.40: Verificación de tipo de la estructura case

Expliquemos un poco más la regla de la figura anterior. La estructura case se puede utilizar para poder elegir una expresión en base al valor de otra. En la estructura, e es la expresión que se evalúa y si su valor coincide con el de algunas de las expresiones e_t el resultado de la expresión será el valor v_t asociado al mismo. Para que una expresión case sea correcta, todas las expresiones internas a la misma deven ser válidas y además todas las expresiones a la izquierda de las "ramas" (e_t) y la expresión a evaluarse en el *case* tienen que ser del mismo tipo. Además las expresiones a la derecha de las "ramas" (v_t) deben ser todas del mismo tipo también. Estableciendo así el tipo del case entero que es el mismo que el de las expresiones a la derecha de las "ramas".

3.2 Semántica

La semántica de InterfLan está dada por dos partes: la semántica declarativa que establece qué datos de entrada son aceptados por un formulario, y por la semántica operacional que modela el comportamiento del formulario frente a las diferentes acciones del usuario sobre él.

3.2.1 Semántica Declarativa

Las semánticas declarativas o descriptivas ilustran la función de cada elemento de la sintaxis de un lenguaje, en contraposición a lo que hace ésta última que solamente define qué elementos lo conforma. En concreto, para InterfLan, su semántica declarativa expresa para un formulario qué valores de entrada son aceptables. Para ello, es necesario entender, de forma abstracta, cómo es un formulario de forma de poder mostrar las restricciones que se deben cumplir para que este formulario sea válido.

Pensemos en un formulario como una estructura que contiene los campos acompañados del tipo de datos que acepta.

$$< c_1 : \alpha_1;\ c_2 : \alpha_2;\ \ldots\ ;\ c_n : \alpha_n >$$

Figura 3.41: Representación del formulario de n campos.

Además, pensemos en que todas las validaciones sobre los campos, las condiciones de obligatoriedad, habilitaciones y cálculo, son listas a nivel global del formulario que recolectan todas las restricciones que fueron especificadas por el desarrollador en el código. En éste las restricciones están escritas a nivel de cada campo solamente por conveniencia a la hora de la tarea de la compilación y ayuda al desarrollador y sus interesados a leer rápidamente el código porque toda la información sobre un campo se encuentra en el mismo. Pero la realidad es que esas restricciones forman parte de la especificación del formulario en general por lo que organizar las restricciones de forma global puede resultar natural también.

$$\begin{array}{cccc} O_1 & E_1 & V_1 & c_{i1} \doteq f_1 \\ O_2 & E_2 & V_2 & c_{i2} \doteq f_2 \\ \vdots & \vdots & \vdots & \vdots \\ O_n & E_n & V_k & c_{im} \doteq f_m \end{array}$$

Figura 3.42: Listado de restricciones del formulario.

Debemos notar que las listas de condiciones de obligatoriedad (O) y de habilitación (E) contienen exactamente la misma cantidad de condiciones que la cantidad de campos del formulario, ya que éstas dos son características de todos los campos por más que no hayan sido específicamente denotadas en el código. No pasa lo mismo con las validaciones de datos (V), de las que pueden existir más de una por campo por lo que podemos decir que la lista de validaciones de datos es de largo k donde k puede ser 0, menor o igual a n o incluso mayor a n. También contamos con el listado de los campos definidos a través de una función. Son aquellos que calculan su valor a partir de una expresión que puede utilizar el valor de otro campo. Estos pueden ser campos `label`, definidos exactamente con una función para calcular su valor, o `input` que se encuentren en una relación de equidad. Entonces, como restricción se listan las definiciones de los m campos que poseen esta restricción.

Una vez establecida una representación para el formulario, debemos expresar las entradas de datos en los campos que son aceptadas teniendo en cuenta todas las restricciones explicadas anteriormente. Es importante

destacar que, durante la entrada de datos permitiremos un nuevo valor para todos los tipos que significa la ausencia de un valor o el borrado del valor ingresado anteriormente; a este valor lo llamaremos `null`. Básicamente, una entrada es aceptada si cumple con cuatro condiciones relacionadas a las listas de restricciones del formulario.

$$< c_1 = a_1;\ c_2 = a_2;\ \ldots\ ;\ c_n = a_n >$$

Figura 3.43: Ingreso de datos a los campos del formulario.

Entonces, dada una entrada de datos al formulario expresada de la siguiente manera donde por cada campo hay un nuevo valor del tipo de datos (α_k), se deberá cumplir que:

- Todas las validaciones de datos expresadas en el formulario al sustituir en la expresión por el nuevo valor de alguno de los campos, evalúe a `True`.

$$(\forall j \in [1..k])\ V_j[\overline{c_i := a_i}] = True$$

Figura 3.44: Cumplimiento de validaciones.

- Si alguna condición de obligatoriedad sobre algún campo resulta ser verdadera (evaluada de la misma forma que la condición anterior), entonces el nuevo valor para cierto campo i debe ser distinto de `null`.

$$(\forall j \in [1..n])\ O_j[\overline{c_i := a_i}] = True \Rightarrow a_i \neq null$$

Figura 3.45: Cumplimiento de obligatoriedad.

- Al igual que sucede con las condiciones de obligatoriedad, en este caso se verifica que si un campo no está habilitado, entonces el valor del mismo debe ser `null`.

$$(\forall j \in [1..n])\ E_j[\overline{c_i := a_i}] \neq True \Rightarrow a_i = null$$

Figura 3.46: Cumplimiento de habilitación.

- Por último se encuentran los cálculos de valores según otros campos o expresiones. No es una condición en sí, pero es importante mencionar que al cambiar la entrada de datos del formulario aquellos campos que tengan funciones de cálculo deberán actualizarse mediante la aplicación de la función que se encuentra en la lista de restricciones y cálculos del formulario.

$$(\forall j \in [1..m])\ c_{ij} = f_j[\overline{c_i := a_i}]$$

Figura 3.47: Cálculo de valor en el campo.

3.2.2 Semántica Operacional

La semántica operacional define la interacción del formulario con las acciones del usuario. En InterfLan el usuario puede interactuar con el formulario solamente de dos formas: mediante el ingreso de un dato en un campo, o enviando el formulario completado (*submit*).

Para la definición de esta semántica sólo nos interesará saber cuál es el estado del formulario en determinando momento, por cuáles estados puede pasar y cuáles son las transiciones entre estos estados. Esto último ya lo hemos mencionado, son las acciones que llegan desde el exterior del formulario (en este caso, desde el usuario) que modifican el estado del formulario.

Antes de centrarnos en las transiciones, definamos los estados por los cuales puede pasar el formulario. El estado del formulario está compuesto por los estados de cada campo del mismo y éstos son los siguientes:

- ***Pristine*:** Un campo prístino significa que sobre el mismo nunca se ha recibido entrada de datos alguna, por lo que todavía no se ha podido determinar su validez. Este estado es útil para que, en los casos en que el formulario nunca fue “tocado" no se ejecuten validaciones innecesarias que frustren al usuario con muchos mensajes de error.

- ***Dirty*:** Un campo “sucio" es aquel el cual ha recibido una entrada de datos pero todavía no se ejecutaron las validaciones sobre el mismo. Este estado suele ser interno al lenguaje ya que el período de tiempo que un campo permanece en estado *dirty* suele ser muy pequeño ya que inmediatamente se ejecutan las validaciones pertinentes.

- ***Valid*:** Un campo se considera válido cuando el valor del mismo cumple con todas las validaciones de dicho campo.

- ***Invalid*:** Un campo se considera inválido cuando el valor del mismo no cumple con alguna de las validaciones de dicho campo.

Entonces, dado los estados de cada campo, el estado del formulario será: válido o prístino, si todos los campos son *valid* o *pristine*, e inválido o "sucio" si al menos contiene un campo en estado *invalid* o *dirty*.

Definidos los estados posibles del formulario y sus campos, podremos centrarnos en las transiciones que existen entre ellos. Primero comencemos con la transición más frecuente en el formulario que es el ingreso de un dato en un campo. Podemos representar esta transición como el pasaje entre un estado cualquiera del formulario y otro resultante de la ejecución de esta transición. La transición es ejecutada cuando un campo cambia de valor por el ingreso de datos con las únicas pre-condiciones de que el campo a cambiar su valor se encuentre habilitado y que el nuevo valor sea del tipo de datos del campo en cuestión [1].

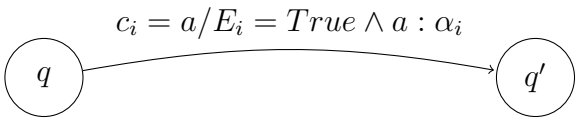

Figura 3.48: Transición al ingresar un dato.

Durante la transición, se marca el campo como *dirty* y se ejecutan todas las validaciones de datos globales del formulario en forma secuencial, además también se vuelven a recalcular las condiciones de habilitación y obligatoriedad para que al finalizar la transición el estado del formulario quede actualizado completamente. Si durante la transición alguna validación de datos no evalúa a *True*, entonces el campo relacionado en esa validación quedará marcado como *invalid* y por lo tanto el formulario también será inválido.

Luego de que el usuario realizó varios ingresos de datos en los campos del formulario, puede estar pronto para enviar el mismo (lo que comúnmente conocemos en Internet como *submit*). En este caso podemos expresar la transición como el paso de un estado del formulario a un estado de aceptación que signifique que el formulario fue enviado correctamente. Al estar todos los campos validados mediante la transición anterior, o sea que ya todos los campos saben si son válidos o no, al momento de enviar el formulario solo basta con chequear que todos los campos sean válidos. En caso de que

[1] Para representar esto esquemáticamente usaremos la misma notación para los campos y restricciones que la de la sección de Semántica Declarativa.

haya algún campo en estado *pristine*, se deberá ejecutar sus condiciones de validación para cerciorarse si ese campo es válido aunque su valor sea `null`.

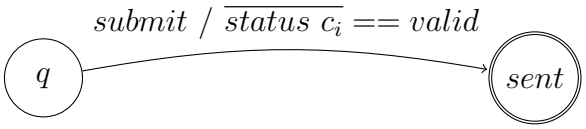

Figura 3.49: Transición al enviar el formulario.

4 Proceso de Desarrollo

En este capítulo repasaremos las tres secciones que componen el compilador de InterfLan. Particularmente, haremos hincapié en la implementación del generador de código ya que la dedicación principal de este proyecto fue puesta en esta sección.

La implementación de todo el lenguaje se ha basado en el mismo esquema que se realizó el curso de "Lenguajes de Programación" de este título junto al libro *"Implementing Programming Languages"* [1] donde se proveen herramientas y fundamentos para el diseño e implementación de un lenguaje.

4.1 Gramática

Como uno de los objetivos de InterfLan es que escribir código en el lenguaje no sea tan diferente a la especificación narrada en lenguaje natural, se decidió basarse ampliamente en la sintaxis de *Ruby* la cual presenta una forma de leer código muy similar a la de leer un texto en prosa. Es un lenguaje en el cual es excepcional encontrar algún símbolo (salvo algunas operadores) porque se prefiere que las operaciones y declaraciones se expresen con palabras para darle más expresividad al código en cuanto a la lectura.

Como se observa en la **Figura 4.1** la gramática del lenguaje está dividida en cuatro niveles:

- **Formulario:** Es la definición del programa en sí. Todos los campos solamente se podrán definir si se encuentran dentro de un formulario. Entonces es aquí donde se definen todos los campos que contendrá el mismo. También a este nivel se puede establecer un nombre al formulario y un título que se establecerá en la etapa de generación de código.

- **Definiciones:** Una vez definido un formulario, encontraremos las definiciones de campos, funciones, enumerados y relaciones. Los campos son los elementos en donde el usuario hará la entrada de datos (`input` y `select`) o donde se mostrarán datos para el caso de las etiquetas

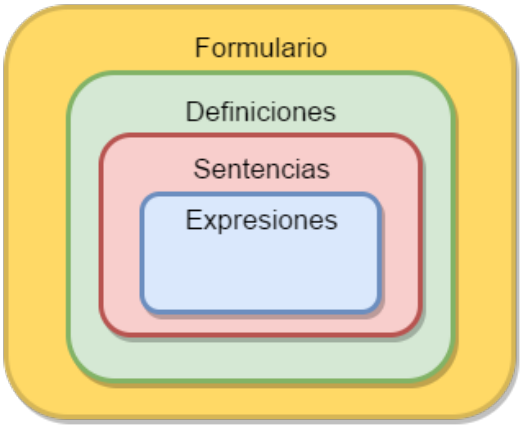

Figura 4.1: Estructura de la gramática de InterfLan.

(`label`). La definición de funciones provee al desarrollador la posibilidad de crear validaciones, cálculos y transformaciones más complejas a las que permiten las expresiones de InterfLan. Todas las definiciones están acompañadas de su tipo excepto los enumerados que constituyen un tipo nuevo. La relación que se especifica a nivel de definición es `equity` debido a que tiene que relacionar específicamente dos campos del formulario para denotar firmemente la relación de circularidad.

- **Sentencias:** Éstas denotan las restricciones que tendrá cada uno de los campos del formulario. Abarcan tanto validaciones de datos, habilitación de un campo y el cálculo de su valor en base a otro componente. Cada campo podrá contener desde ninguna restricción a todas las que quiera expresar el desarrollador; esto se puede realizar o bien escribiendo hasta una restricción a continuación de la definición del campo, o dentro de un bloque `let...end`.

```
input nombre:string ;
```

Código 4.1: Control sin sentencias.

```
input nombre:string not_empty ;
```

Código 4.2: Control con una sentencia.

```
input "Fecha de Nacimiento" as fecha_nac:datetime let
    not_empty ;
    check (self <= now) ;
```

```
end
```

Código 4.3: Control con más de una sentencia.

- **Expresiones:** Son los átomos del lenguaje, forman los valores ya sea desde un literal como un *string* ("*hola*"), otro campo identificado por su nombre de variable (`name`), una operación, una aplicación de una función, etc. Todas las expresiones poseen un nivel de precedencia que indica qué expresiones se tienen que evaluar antes que otras, lo cual ayudan a que la sintaxis no se "ensucie" mediante el uso excesivo de paréntesis. Las expresiones de nivel más alto son aquellas que no necesitan evaluarse o que se necesitan evaluar primero y las de nivel más bajo son aquellas que se evalúan al final.

Luego de especificada la gramática, se utilizó la herramienta *BNFC* [12] (*BNF Converter*) la cual produce *Lexer*, *Parser* y sintaxis abstracta en *Haskell* a partir de una gramática especificada en *BNF*.

4.2 *Type Checker*

Una vez que se obtuvo la gramática formalizada, se debió realizar los procedimientos de inferencia de tipo de las expresiones para comprobar que los tipos de las mismas concuerdan con las permitidas en las sentencias y definiciones utilizadas. Esto se debe a que en las etapas anteriores de *Lexer* y *Parser* sólo se verifican errores de sintaxis. En esta etapa nos podremos encontrar con errores de bajo nivel de abstracción, como puede ser una expresión entera donde se necesita una booleana; y otros de alto nivel como una circularidad en dependencias entre campos. De manera similar a como se procedió en el curso de Lenguajes de Programación, se dispuso a analizar qué cosas se necesitaban para verificar estos posibles errores. Para ello se precisó la generación de contextos, antes de comenzar la verificación del código, que proveerán más información acerca de las definiciones y las dependencias que se generan entre los campos. Se concluye entonces que el compilador, en esta etapa, necesitará pasar dos veces por el código para realizar todas las validaciones pertinentes: una para generar los contextos que se describirán a continuación y otra para realizar la inferencia de tipos de las expresiones y que los mismos coinciden con las sentencias y definiciones.

4.2.1 Contextos

Los contextos necesarios para poder verificar las expresiones y sentencias son los siguientes:

- **Contexto de campos:** Éste es una tabla con el nombre de los campos definidos en el formulario y su tipo. Es importante para verificar que cuando se haga referencia a otro campo, se pueda verificar que el mismo exista y que el tipo del mismo coincida en la expresión donde se está utilizando.
- **Contexto de funciones:** Éste es una tabla que contiene el nombre de las funciones definidas en el formulario, acompañados de la firma de la misma, o sea, el tipo de retorno de la función y los tipos de los parámetros que recibe. La información recolectada en este contexto se utiliza para verificar que, cuando se hace una aplicación de una función, la misma se encuentre definida en el formulario, los parámetros recibidos coincidan con los tipos de los parámetros y que el retorno de la función coincida en la expresión donde se está utilizando la aplicación.
- **Contexto de enumerados:** Éste es una tabla que contiene los nombres de los enumerados definidos en el formulario, en conjunto a los valores válidos que permite el mismo. La información que brinda este contexto es fundamental para comprobar que una expresión de este tipo solamente contemple los valores establecidos en el enumerado.
- **Contexto de dependencias:** Debido a que nuestro lenguaje permite dependencias entre los valores de los campos, se necesita verificar que no existan circularidades entre dichas dependencias, ya que la circularidad podría provocar que los campos no se pudieran calcular. La decisión tomada para verificar este problema es mediante la formación de grafos de dependencias en donde, luego de armados, poder verificar las circularidades. Estos grafos contienen como vértices a todos los campos definidos en el formularios y las aristas (que representan las dependencias, leyéndose de la forma "depende de") serán agregadas mediante la recorrida del formulario donde se originan las dependencias, tanto en las sentencias `calculate` como con las definiciones de bloques `equity`. En el último grafo, donde se plasman las relaciones `equity`, se agregan las dependencias `calculate` del resto de los campos, ya que en su naturaleza `equity` es una relación de cálculo pero que solamente permitirá circularidad entre dos campos, por lo que se tiene que verificar que no se forme circularidad entre un campo y el bloque `equity`.

La generación de estos contextos requiere, como se anticipó, de una recorrida completa del código para recolectar todas las definiciones y las relaciones de dependencias. Una vez creados todos los contextos se puede pasar a la última recorrida de código para chequear los tipos ayudándonos de la información recolectada.

4.2.2 Verificación de tipos

Con la información de los contextos generados como se explicó anteriormente, se recorre el árbol de sintaxis abstracta producida por el *lexer* y *parser*, agregando al mismo etiquetas de tipo a las expresiones y controlando que los tipos de las expresiones siguen la estructura que se definió. El etiquetado de las expresiones con sus tipos inferidos es útil para la etapa de generación de código.

Analizando con más detalle, se parte de un formulario que debe ser validado y un contexto que contiene toda la información relevante mencionada en la sección anterior. El formulario en un primer nivel consta de definiciones, las mismas pueden ser funciones, enumerados, relaciones `equity`, o campos (`select`, `input` o `label`). Las funciones no tienen validaciones de tipo ya que el cuerpo de la misma no está escrito en código InterfLan, sino que está en *JavaScript*. Es importante destacar que, debido a esto, en el cuerpo de la función no está disponible el acceso a los campos definidos en todo el formulario. Solamente se podrán utilizar los parámetros pasados a la función; por lo cual no se puede generar recursividad ni dependencias de ningún tipo entre funciones. Los enumerados tienen la única validación de que, si se declaran por extensión (con una lista de valores aceptados), sólo se deberá verificar que no existan valores repetidos. Pero si se define mediante una dependencia `domain` se verificará que cada uno de los enumerados devueltos por la expresión `case` de esta dependencia, sean correctos. Las relaciones `equity` tienen como único cometido denotar fuertemente la relación circular de cálculo entre dos campos `input`. Para ello se deberá verificar que ambos campos sean del mismo tipo, y que ambos tengan una sentencia `calculate` que contenga en su expresión al otro campo para que de esa manera se establezca la circularidad.

Llegando a las definiciones más interesantes dentro del formulario, se encuentran los campos. Según la naturaleza de cada tipo de campo, se realizan diferentes validaciones antes de ir directamente a verificar las sentencias que contienen y donde se encuentran las expresiones a chequear. En los campos `select` se verifica que el tipo de este campo sólo puede ser un enumerado, ya que el por su naturaleza fue concebido para la elección de un valor entre varios valores establecidos, y que no contenga alguna relación `calculate`.

En los campos `input` no deberá haber alguna relación `calculate`, a no ser que el campo se encuentre dentro de un bloque `equity`, ya que esa relación se pensó solamente para los campos `label` cuyo objetivo principal es obtener resultados intermedios de otros campos para facilitar validaciones o para mostrar al usuario este resultado que puede llevarlo a tomar decisiones en la entrada del formulario. Luego de estas verificaciones es necesario recorrer las sentencias de los campos y ver que todas ellas sean válidas. Pero con los contextos generados en la etapa anterior no basta, ya que en las expresiones se puede hacer mención del valor del control mediante la palabra `self`, así que se agrega a los contextos (antes de verificar las sentencias de un control) el tipo que tiene `self` en esa sección del código.

Las verificaciones de las sentencias, expresadas de forma narrada son las siguientes:

- `not_empty`, `check` y `enable`:
  - La expresión que las compone debe ser de tipo `bool`.
- `range`:
  - el tipo de ambas expresiones (el extremo inferior y el extremo superior) son del mismo tipo que el `self` y no son `enumerate`.
- `calculate`:
  - El tipo de la expresión que lo compone debe ser igual al del `self`
- `regex`: No tiene verificación específica ya que está compuesta por un *string*.

Para culminar, el proceso para las expresiones es similar, sólo que cuando se infiere el tipo de una expresión, agregamos una estructura al árbol con el tipo de la expresión y la expresión en sí (el etiquetado de expresiones que tratamos al principio de la sección). Las únicas reglas interesantes para la inferencia de tipos son las operaciones aritméticas que están sobrecargadas para los `integer` y los `float`, y la suma que además también está disponible para `string` (permitiendo la concatenación).

4.2.3 Validación de los grafos de dependencias

En la etapa final de la verificación del formulario, se buscan errores en cuanto a dependencias en los grafos generados en la segunda “pasada" del *type*

checker. Una de las tareas más difíciles de todo este trabajo fue determinar cuáles eran las condiciones que debían tener los grafos que se generaron en el contexto. En el trabajo anterior se entendía que todas las dependencias sobre los campos no deberían contener circularidades. Ahora las validaciones de datos, habilitaciones y criterios de obligatoriedad se verifican de forma lineal y no simultánea, no hay forma de que el formulario quede en *loop* para determinar una de estas características, por lo que el chequeo de circularidad de estas dependencias ya no era necesario.

Para las dependencias de cálculo y equidad sí es necesario establecer una condición para evitar circularidades ya que para nosotros es más crítico que no se produzcan inconsistencias a nivel de los datos que contienen los campos. Para ello, se comenzó a analizar cuándo un formulario era correcto y cuándo no. Fundamentalmente, la restricción en las dependencias de `calculate` es que no existan circularidades entre ellas para que de esta manera se eviten inconsistencias en los estados de estos a causa de la circularidad cuando el usuario ingrese datos en estos campos. Solamente existe una excepción a esta regla que es la definición de `equity` ya que explícitamente se está definiendo una dependencia circular de tipo `calculate` entre dos campos de un mismo tipo de datos. Ahora bien, de los grafos generados en el contexto anteriormente, el grafo correspondiente contiene también las dependencias de otros `calculate` debido a que no se podría generar una circularidad entre uno de los campos participantes en la relación `equity` y otro campo por fuera de esta relación.

En la primera versión de InterfLan, se habían detectado este tipo de problemas entre las dependencias por lo que se había tomado la decisión de agregar las dependencias `calculate` al grafo que contiene las relaciones `equity`. Pero el estudio de este conflicto quedó inconcluso; luego de finalizado el trabajo se encontraron formularios que el *type checker* no validaba cuando en la realidad no se estaba incurriendo en una circularidad en particular. Este estudio trataba de un análisis entre los grados de entrada y salida de los vértices del grafo de dependencias, pero lo que no fue tomado en cuenta es la cantidad de vértices que podrían estar relacionados. Siempre se trabajó pensando en que la relación conflictiva se daba entre tres campos (2 en `equity` y 1 en `calculate` con los demás).

De hecho, el problema que se mantenía entre estas relaciones era el mismo que el resto de los otros tipos de dependencias: encontrar las circularidades no deseadas, salvo que en este caso hay que ignorar las circularidades generadas específicamente por la relación `equity`, tomando como punto de partida que la verificación de los campos en circularidad se realiza a través del algoritmo de la búsqueda de las componentes fuertemente conexas, donde se obtiene cada componente indicando si la misma es cíclica o no. De este modo lo

único que queremos permitir en la relación `equity` es que hayan componentes conexas cíclicas entre 2 vértices, ya que el resto de los campos en el grafo de dependencias analizados debería pertenecer a otra componente conexa.

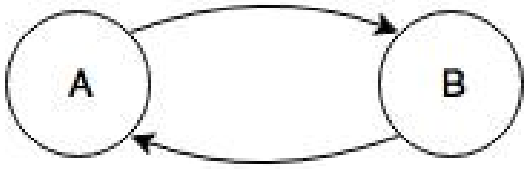

Figura 4.2: Grafo relación de `equity`

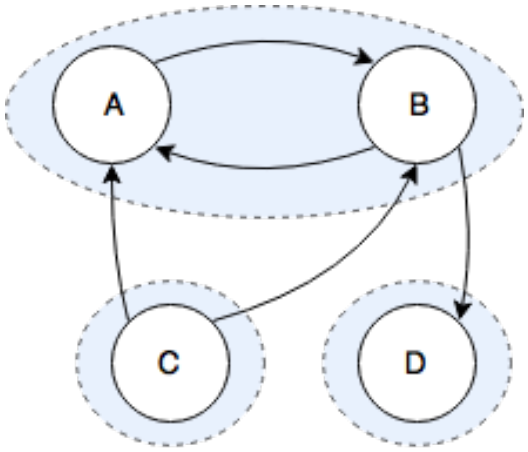

Figura 4.3: Ejemplo de dependencias válidas

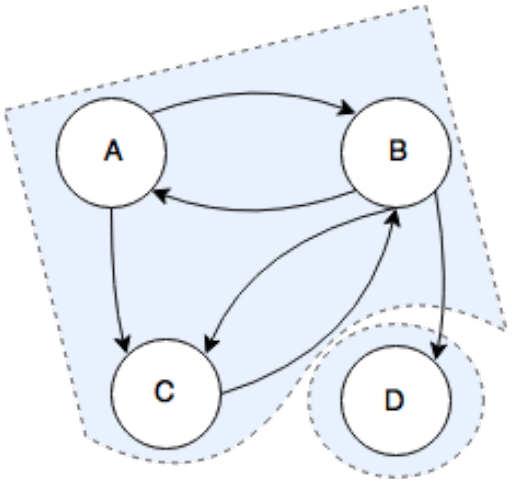

Figura 4.4: Ejemplo de dependencias inválidas

4.3 Generador de Código

Una vez pasadas las etapas anteriores donde se verificaron todas las restricciones del lenguaje, desde las sintácticas hasta las de dependencias entre campos, es cuando finalmente se debe compilar el código.

A lo largo de esta sección se comenta qué herramientas fueron utilizadas para realizar este proceso y se reafirma por qué es más sencillo especificar un formulario en InterfLan y no directamente con las herramientas utilizadas.

4.3.1 Modularización de paquetes

Antes de comenzar a implementar el generador de código en sí, se decidió resolver el hecho de que por cada etapa nueva del compilador se tenían que importar todos los módulos de la etapa anterior, ya que el compilador está formado por tres etapas una a continuación de la otra. Para ello, se decidió modularizar cada etapa en un paquete de *Cabal* [13] que permite de forma más sencilla importar todos los módulos de un paquete simplemente expresando la dependencia a éste.

Cabal es un sistema de paquetes de *Haskell* el cuál permite configurar, compilar y distribuir paquetes de forma sencilla. Uno puede crear un paquete del tipo *library* donde este paquete solamente va a dejar disponibles ciertos módulos de *Haskell* a quien lo importe; o se puede crear un paquete del tipo *executable* el cual contiene un módulo *Main* que se deja disponible al usuario mediante la llamada por el comando del paquete. Justamente este último

tipo es el que se usará para contener al generador de código, ya que en el mismo importaremos los paquetes de las etapas anterior y una vez que el usuario instale el paquete en su computadora tendrá el comando `interflan` disponible que es el punto de entrada al compilador. Bastará pasarle al comando `interflan` un archivo `.ifl` y se obtendrá como resultado la compilación del mismo.

4.3.2 Herramientas utilizadas

Para realizar la generación de código se analizó en profundidad las herramientas de *JavaScript* vistas en la **Sección 1.3.4** (*ReactJS*, *Redux* y *Redux Form*). Estas herramientas permiten la definición de componentes *HTML* con comportamiento agregado, por lo cual la forma de escritura entre nuestro lenguaje y estas herramientas se vuelve similar. Se comenzó por hacer algunas pruebas de concepto, utilizando todas estas herramientas para determinar cuáles eran útiles para esta tarea. Para ello, se tomó como punto de partida el formulario definido en el ejemplo anterior del **Capítulo 2**, y se realizó una compilación "a mano" de este formulario utilizando las funcionalidades de *Redux Form* ya que en principio su dominio tan específico prometía ser la solución a la generación de código.

En *Redux Form* ya está definido el componente que encapsula el comportamiento de un campo de un formulario, en ese caso solo bastaba con dar una función de *render* (que define como se ve el campo hacia el usuario) y las validaciones específicas del campo. El problema de depender de esta herramienta para el manejo del estado, es que en ciertas restricciones circulares, el formulario a veces quedaba en *loop* debido a su implementación, y porque otras restricciones también no resultaban fácilmente expresables en ella, por lo que *Redux Form* dejó de ser tan prometedora como herramienta porque no era fácilmente adaptable al funcionamiento de la semántica de nuestro lenguaje.

Viendo este resultado fue que se decidió utilizar solamente *ReactJS*, que nos brinda la posibilidad de la definición de componentes con el comportamiento que uno decida. Entonces se crearon los componentes que encapsulan el comportamiento del ingreso de datos en un campo definido por la semántica de nuestro lenguaje, y las verificaciones globales al formulario. Igualmente para la creación de los componentes que definen los campos, se tomó inspiración de *Redux Form* en su definición pero con el comportamiento necesario por el lenguaje.

Entonces, visto todo esto, traducir un campo `input` desde InterfLan a *ReactJS* con nuestros componentes específicos resulta bastante sencillo como podemos observar en los siguientes códigos.

```
...
label "Edad" as edad:integer let
    calculate (calcular_edad(f_nacimiento)) ;
    check (self >= 18) ;
end
...
```

Código 4.4: Campo de ejemplo para compilar

```
...
<LabelField
    type='number' step='1'
    name='edad' label='Edad'
    validations={[checkValidation((value) => (value >= 18), 'Debe
        ser mayor a 18.')]}
    calculate={calcular_edad(this.state.f_nacimiento)}
/>
...
```

Código 4.5: Campo de ejemplo compilado

La traducción es casi inmediata, salvo por el detalle de qué componente de *HTML* es `LabelField`. Pues no es ninguno dentro de *HTML*, sino que es un componente de *ReactJS* que encapsula cómo se tiene que mostrar un campo de estos (los *tags* de *HTML* necesarios para que el usuario vea el campo) y el comportamiento del campo, en esto último nos detendremos un poco más. Cada campo, aparte de expresar como se tiene que ver en la interfaz, mantiene su estado (valor ingresado y estado de validez) para que, al momento de recibir un cambio en el valor del mismo, sea posible ejecutar todas las validaciones y determinar el nuevo estado del campo.

4.4 Resultados de experimentación

Para la experimentación con el compilador completo se probaron dos ejemplos: el que se desarrolló en el capítulo de la guía del lenguaje y el planteado en el proyecto de Degrandi [2] sobre la interfaz de programación marcapasos. En este ejemplo, las validaciones resultaban más complejas, ya que los valores para algunos campos debían cumplir ciertos rangos y también ciertos campos debían cumplir con relaciones de cálculo entre ellos. Las especificaciones y el código de este ejemplo se encuentra disponible en el **Anexo B.2**.

Además, en la presentación del proyecto se agregó otro ejemplo sobre la implementación de un formulario de censo de población disponible en el **Anexo B.3**.

5 Conclusiones y trabajo a futuro

De este trabajo podemos concluir que hemos logrado completar el diseño del lenguaje mediante la definición de una semántica formal expresando qué entradas son aceptadas por el formulario y cómo interactúa el formulario con las acciones del usuario. Se ha conseguido tener una herramienta usable para compilar el lenguaje a *HTML* y *JavaScript* y asimismo se ha probado que el compilador con dos ejemplos claves, uno que contiene la mayoría de las restricciones que se pueden expresar en InterfLan y otro que define una formulario complejo en su especificación como es la interfaz de programación de los marcapasos. Con respecto al tratamiento de las restricciones de las dependencias que se plantean en los objetivos del trabajo, se encontró una solución simple y razonablemente completa donde solamente se verifican las circularidades entre las dependencias `calculate` y `equity`, mediante el algoritmo de componentes fuertemente conexas visto en la **sección 4.2.3**.

A futuro sería interesante ser menos restrictivo en el tratamiento de las dependencias circulares de `equity` ya que hasta el momento no se está permitiendo encadenar este tipo de dependencias debido que dentro de las interfaces analizadas (formularios web) no se han encontrado ejemplos concretos que justifiquen la generalización de esta dependencia. También sería interesante implementar mecanismos para definir tipos enumerados en base a información externa como podría ser obtener el listado de valores de una base de datos o desde una API.

6 Referencias Bibliográficas

[1] A. Ranta, *Implementing Programming Languages.* Londres, Reino Unido: College Publications, 2012.

[2] S. Degrandi, "Herramientas de especificación y generación de módulos de entrada y validación de datos," PEDECIBA Informática - InCo, Facultad de Ingeniería - UdeLaR, Montevideo, Uruguay, 2005.

[3] J. Helms, *User Interface Markup Language (UIML) Version 4.0.* Oasis, 2008.

[4] Google, "AngularJS API Docs." [Online]. Available: https://docs.angularjs.org/api [Accessed: 12-Jun-2017].

[5] "Knockout: Introduction." [Online]. Available: http://knockoutjs.com/documentation/introduction.html [Accessed: 12-Jun-2017].

[6] Facebook, "React: A JavaScript library for building user interfaces." [Online]. Available: https://reactjs.org [Accessed: 15-Sep-2017].

[7] "Redux." [Online]. Available: https://redux.js.org [Accessed: 15-Sep-2017].

[8] "Redux Form - Getting Started." [Online]. Available: https://redux-form.com/7.2.3/docs/gettingstarted.md/ [Accessed: 15-Sep-2017].

[9] "Ruby on Rails: Guides." [Online]. Available: http://guides.rubyonrails.org/ [Accessed: 12-Jun-2017].

[10] "Django Documentation." [Online]. Available: https://docs.djangoproject.com/en/1.10/ [Accessed: 12-Jun-2017].

[11] E. Santurio and J. Silveira, "Interflan: Interface language," Facultad de Ingeniería, Universidad ORT Uruguay, Montevideo, Uruguay, 2017.

[12] M. Forsberg and A. Ranta, "The Labelled BNF Grammar Formalism," Department of Cumputing Science - Chalmers University of Technology and the University of Gothenburg, Sweden, SE-412 96, February 2005.

[13] "Cabal User Guide." [Online]. Available: https://www.haskell.org/cabal/users-guide/ [Accessed: 15-Sep-2017].

[14] Instituto Nacional de Estadística, "Censos 2011: Cuestionarios y planillas especiales." [Online]. Available: http://www.ine.gub.uy/censos-2011 [Accessed: 05-Abr-2018].

Anexos

A Gramática

This section was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language.

The lexical structure of Interflan

Literals

String literals *String* have the form `"`x`"`, where x is any sequence of any characters except `"` unless preceded by \.

Integer literals *Integer* are nonempty sequences of digits.

Double-precision float literals *Double* have the structure indicated by the regular expression `digit+ '.' digit+ ('e' ('-')? digit+)?` i.e.\ two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.

Datetime literals are recognized by the regular expression ‘'"' ["12"] digit digit digit '-' ["01"] digit '-' ["0123"] digit 'T' ["012"] digit ':' ["012345"] digit '"'‘

Id literals are recognized by the regular expression ‘letter (letter | digit | '_')*‘

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in Interflan are the following:

`add_days`	`add_hours`	`add_minutes`	`add_months`
`add_years`	`as`	`bool`	`calculate`
`case`	`check`	`combo`	`datetime`
`domain`	`enable`	`end`	`enum`
`enumerate`	`equity`	`false`	`float`
`form`	`function`	`hide_unless`	`in`
`input`	`integer`	`label`	`let`
`not_empty`	`now`	`of`	`otherwise`
`radio`	`range`	`regex`	`select`
`self`	`string`	`subs_days`	`subs_hours`
`subs_minutes`	`subs_months`	`subs_years`	`true`

The symbols used in Interflan are the following:

:	;	(	)
[	]	,	..
.	!	*	/
%	+	-	<
>	<=	>=	==
!=	&&	||	->

Comments

Single-line comments begin with #.There are no multiple-line comments in the grammar.

The syntactic structure of Interflan

Non-terminals are enclosed between < and >. The symbols -> (production), | (union) and **eps** (empty rule) belong to the BNF notation. All other symbols are terminals.

Program	->	`form` *Name* `let` *[Def]* `end`
Def	->	`input` *Name* `:` *Type Body*
	|	`label` *Name* `:` *Type Body*
	|	`select` *SelectCtrl Name* `:` *Type Body*
	|	`function` *Id* `:` *Type FuncBody*
	|	`enum` *Id EnumBody*
	|	`equity let` *Def Def* `end`
[Def]	->	**eps**
	|	*Def [Def]*

Name -> *Id*
| *String* `as` *Id*

Type -> *PrimType*
| *Id*

Body -> `;`
| *Stm*
| `let` *[Stm]* `end`

FuncBody -> `let` *String* `end`
| `(` *[Param]* `)` `let` *String* `end`

EnumBody -> `[` *[EnumValue]* `]` `;`
| `let` *Stm* `end`

EnumValue -> `(` *String* `,` *Id* `)`

[EnumValue] -> *EnumValue*
| *EnumValue* `,` *[EnumValue]*

Stm -> `not_empty` *NotEmptyBody*
| `regex` *String ValidationBody*
| `range` *Exp* `..` *Exp ValidationBody*
| `check` *Exp ValidationBody*
| `enable` `(` *Exp* `)` `;`
| `calculate` `(` *Exp* `)` `;`
| `domain` `(` *Exp* `)` `;`

[Stm] -> **eps**
| *Stm [Stm]*

NotEmptyBody -> `;`
| *Exp* `;`

ValidationBody -> `;`
| *String* `;`

Exp17 -> `true`
| `false`
| *Integer*
| *Double*
| *String*
| `now`
| *Datetime*
| `self`
| *Id*
| *Id* `.` *Id*
| `[` *[EnumValue]* `]`
| *Id* `in` `[` *[Id]* `]`
| `(` *Exp* `)`

Exp16 -> *Id* `(` *[Exp]* `)`

| | *Exp17*
Exp15 -> `!` *Exp15*
| *Exp16*
Exp14 -> *Exp14* `*` *Exp15*
| *Exp14* `/` *Exp15*
| *Exp14* `%` *Exp15*
| *Exp15*
Exp13 -> *Exp13* `+` *Exp14*
| *Exp13* `-` *Exp14*
| *Exp14*
Exp12 -> *DTOp* `(` *Exp12* `,` *Exp13* `)`
| *Exp13*
Exp11 -> *Exp11* `<` *Exp12*
| *Exp11* `>` *Exp12*
| *Exp11* `<=` *Exp12*
| *Exp11* `>=` *Exp12*
| *Exp12*
Exp10 -> *Exp10* `==` *Exp11*
| *Exp10* `!=` *Exp11*
| *Exp11*
Exp9 -> *Exp9* `&&` *Exp10*
| *Exp10*
Exp7 -> *Exp7* || *Exp9*
| *Exp8*
Exp6 -> `case` *Exp7* `of` *[Branch]* *CaseEnd*
| *Exp7*
CaseEnd -> `end`
| `otherwise ->` *Exp* `end`
Exp -> *Exp1*
Exp1 -> *Exp2*
Exp2 -> *Exp3*
Exp3 -> *Exp4*
Exp4 -> *Exp5*
Exp5 -> *Exp6*
Exp8 -> *Exp9*
[Exp] -> **eps**
| *Exp*
| *Exp* `,` *[Exp]*
Branch -> *Exp -> Exp*
[Branch] -> *Branch*
| *Branch* `;` *[Branch]*

Param	->	*Id* : *Type*
[Param]	->	*Param*
	|	*Param* , *[Param]*
SelectCtrl	->	`combo`
	|	`radio`
PrimType	->	`bool`
	|	`integer`
	|	`float`
	|	`datetime`
	|	`string`
	|	`enumerate`
DTOp	->	`add_years`
	|	`add_months`
	|	`add_days`
	|	`add_hours`
	|	`add_minutes`
	|	`subs_years`
	|	`subs_months`
	|	`subs_days`
	|	`subs_hours`
	|	`subs_minutes`
[Id]	->	*Id*
	|	*Id* , *[Id]*

B Casos de Estudio

B.1 Solicitud de Línea de Crédito

```
form "Solicitud de Línea de Crédito" as linea_de_credito let
    input "Nombre" as nombre:string let
        not_empty ;
    end

    input "Apellido" as apellido:string let
        not_empty ;
    end

    input "Dirección de correo electrónico" as email:string let
        not_empty ;
        regex "(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0
            -9-.]+$)" ;
    end

    function verificar_cedula:bool (cedula:integer) let
        "cuerpo de la función en JavaScript"
    end

    input "Cédula de identidad" as cedula:integer let
        not_empty ;
        check (verificar_cedula(self)) ;
    end

    input "Fecha de Nacimiento" as f_necimiento:datetime let
        not_empty ;
        range (subs_years now 150)..(subs_years now 18) ;
    end

    function calcular_edad:integer (fnac:datetime) let
        "cuerpo de la función en JavaScript"
```

```
end

label "Edad" as edad:integer let
    calculate (calcular_edad(f_nacimiento)) ;
end

 enum estado_civil [("Casado/a",CAS),("Soltero/a",SOL),
                ("Concubino/a",CON),("Viudo/a",VIU),
                ("Divorciado/a", DIV)] ;

select "Estado civil" as ecivil:estado_civil let
    not_empty ;
end

input "Fecha concubinato o matrimonio" as f_mat:datetime let
    not_empty (ecivil in [CAS,CON]) ;
    enable (ecivil in [CAS,CON]) ;
    check (edad >= 14) ;
end

enum paises [("Uruguay",UY), ("Brasil", BR), ("Argentina", AR)] ;

select "País de nacimiento" as p_nacimiento:paises let
    not_empty ;
end

enum ciudades let
    domain (case p_nacimiento of
                paises.UY -> [("Montevideo", MVD), ("Rocha", ROC),
                           ("Canelones", CAN)] ;
                paises.BR -> [("Brasilia", BRS),("Fortaleza", FRT),
                           ("Recife",RCF)] ;
                paises.AR -> [("Buenos Aires", BAS), ("Santa Fe",STF
                    ),
                           ("Córdoba", CBA)]
           end) ;
end

select "Ciudad de nacimiento" as c_nacimiento:ciudades let
    not_empty ;
end

equity let
```

```
        input "Mínimo ingreso nominal" as ingresos:integer let
            not_empty ;
            calculate (credito / 2) ;
        end

        input "Máximo línea de crédito" as credito:integer let
            not_empty ;
            calculate (ingresos * 2) ;
        end
    end
end
```

Código B.1: Formulario completo.

B.2 Formulario de marcapasos

En este anexo se muestra el formulario completo de la especificación de la interfaz de programación de marcapasos formulada por Degrandi [2].

Especificación del formulario

En la siguiente tabla se puede observar la especificación de los datos de la interfaz de programación de un marcapasos bicameral.

Parámetro	**Rango de Valores**
Modos	DDD, VDD, VVI, VVT, VOO, OVO, AAI, AAT, AOO, OAO, OOO
Frecuencia	32 a 120 ppm (46 valores)
Frecuencia Máxima	60 a 150 ppm (46 valores)
AV post estímulo	63 a 281 ms (15 valores)
AV post sensado	53 a 272 ms (15 valores)
Blanking	21 a 68 ms (4 valores)
Amplitud de pulso	2.5 V y 5.0 V
Ancho de pulso	122 a 1464 μs (12 valores)
Refractario	125 a 430 ms (16 valores)
Sensibilidad Auricular	0.5 a 4.0 mV (8 valores)
Sensibilidad Ventricular	1.0 a 8.0 mV (8 valores)
Porcentaje de histéresis	0% a 20%
Polaridad de estímulo	Unipolar/Bipolar
Polaridad de sensado	Unipolar/Bipolar

Tabla B.1: Especificación de datos para programar un marcapasos bicameral

Además, de los parámetros numéricos se menciona una serie de datos adicionales que el formulario deberá verificar dependiendo del ingreso de ciertos datos.

Por ejemplo, se conoce que los modos de funcionamiento de un marcapasos puede ser:

- Modos unicamerales ventriculares
- Modos unicamerales auriculares

- Modos bicamerales
- Modo apagado

Además, todos estos modos se clasifican en:

- Modo de sensado de la actividad cardíaca
- Modo de estimulación cardíaca
- Modo de sensado y estimulación cardíaca

Luego de seleccionado un modo, dependiendo de qué tipo de modo se seleccionó, ya sea de sensado, estimulación o ambos, se deben utilizar los siguientes campos:

Si se seleccionó un modo de sensado se deben especificar los campos:

- Refractario
- Sensibilidad
- Polaridad del sensado

Si se seleccionó un modo de estimulación se deben especificar los campos:

- Frecuencia y período
- Ancho del pulso
- Amplitud del pulso
- Polaridad del pulso

Aparte de estas restricciones en cuanto al modo de funcionamiento seleccionado existen otras con respecto a los valores ingresados en los demás parámetros. Algunas son propiedades que se deben cumplir entre dos campos como son el caso de la frecuencia y el período que una es inversa de la otra, y las demás son validaciones de contralor, como lo son las siguientes:

- Los refractarios deben ser menores que el período.
- Los refractarios deben ser menores que el período de *trigger*.
- El período de *trigger* debe ser menor que el período.
- El período de la frecuencia máxima debe ser menor que el período.
- La suma de cualquiera de los refractarios más el período AV debe ser menor que el período de la frecuencia máxima.
- El *blanking* debe ser menor que los períodos AV.

Implementación del formulario

```
form pacemaker let

  enum pm_mode [("VOO", VOO),
    ("OVO", OVO),
    ("VVI", VVI),
    ("VVT", VVT),
    ("AOO", AOO),
    ("OAO", OAO),
    ("AAI", AAI),
    ("AAT", AAT),
    ("VDD", VDD),
    ("DDD", DDD),
    ("OOO", OOO)] ;

  enum polarity [("Unipolar", UNI), ("Bipolar", BIP)] ;

  enum amplitude_range [("2.5 V", V2_5), ("5.0 V", V5_0)] ;

  select mode:pm_mode not_empty;

  equity let
    input rate:integer let
      enable (mode != pm_mode.OOO);
      range 32..120 "La frecuencia debe estar entre 32 y 120.";
      calculate (60000/period);
    end

    input period:integer let
      enable (mode != pm_mode.OOO);
      calculate (60000/rate);
    end
  end

  equity let
    input upper_rate:integer let
      enable (mode != pm_mode.OOO);
      range 60..150 "La frecuencia máxima debe estar entre 60 y 150.";
      calculate (60000/upper_period);
    end

    input upper_period:integer let
```

```
    enable (mode != pm_mode.OOO);
    calculate (60000/upper_rate) ;
    check (self < period) "El período correspondiente a la frecuencia má
        xima debe ser menor que el período" ;
  end
end

equity let
  input trigger_rate:integer let
    enable (mode in [VVT, AAT]);
    calculate (60000/trigger_period);
  end

  input trigger_period:integer let
    enable (mode in [VVT, AAT]);
    calculate (60000/trigger_rate) ;
  end
end

input hyst_rate:integer;

label hyst_percentage:integer let
  calculate (100 * (rate - hyst_rate)/rate);
  check (self <= 20) "El porcentaje de histéresis debe ser menor o igual a
      20%";
  check (self >= 0) "El porcentaje de histéresis debe ser mayor o igual a
     0%";
end

input pace_av:integer let
  enable (mode == pm_mode.DDD);
  range 63..281 "El valor debe estar entre 63 y 281.";
end

input sense_av:integer let
  enable (mode in [VDD, DDD]);
  range 53..272 "El valor debe estar entre 53 y 272.";
end

input blanking:integer let
  enable (mode == pm_mode.DDD);
  range 21..68 "El blanking debe estar entre 21 y 68.";
  check (self < pace_av) "El blanking debe ser menor que período AV de
```

```
        estimulación";
    check (self < sense_av) "El blanking debe ser menor que período AV de
        sensado";
  end

  input anti_pmt:bool not_empty;

# Cámara Aurícula
  select atrium_amplitude:amplitude_range let
    enable (mode in [AOO, AAI, AAT, DDD]);
  end

  input atrium_pulseWidth:integer let
    enable (mode in [AOO, AAI, AAT, DDD]);
    range 122..1464 "El valor debe estar entre 122 y 1464.";
  end

  select atrium_pacePolarity:polarity let
    enable (mode in [AOO, AAI, AAT, DDD]);
  end

  input atrium_sensitivity:float let
    enable (mode in [OAO, AAI, AAT, VDD, DDD]);
    range 0.5..4.0 "El valor debe estar entre 0.5 y 4.0.";
  end

  select atrium_sensePolarity:polarity let
    enable (mode in [OAO, AAI, AAT, VDD, DDD]);
  end

  input atrium_refractory:integer let
    enable (mode in [OAO, AAI, AAT, VDD, DDD]);
    range 195..430 "El valor debe estar entre 195 y 430.";
    check (self < period) "El refractario auricular debe ser menor que el
        período";
    check (self < trigger_period) "El refractario auricular debe ser menor
        que el período de trigger";
    check (self + pace_av < upper_period) "La suma del refractario
        auricular más el período AV de estimulación debe ser menor que el perí
        odo correspondiente a la frecuencia máxima";
    check (self + sense_av < upper_period) "La suma del refractario
        auricular más el período AV de sensado debe ser menor que el período
        correspondiente a la frecuencia máxima";
```

```
  end


# Cámara Ventrícula
  select ventricle_amplitude:amplitude_range let
    enable (mode in [VOO, VVI, VVT, VDD, DDD]);
  end

  input ventricle_pulseWidth:integer let
    enable (mode in [VOO, VVI, VVT, VDD, DDD]);
    range 122..1464 "El valor debe estar entre 122 y 1464.";
  end

  select ventricle_pacePolarity:polarity let
    enable (mode in [VOO, VVI, VVT, VDD, DDD]);
  end

  input ventricle_sensitivity:float let
    enable (mode in [OVO, VVI, VVT, VDD, DDD]);
    range 1.0..8.0 "El valor debe estar entre 1.0 y 8.0.";
  end

  select ventricle_sensePolarity:polarity let
    enable (mode in [OVO, VVI, VVT, VDD, DDD]);
  end

  input ventricle_refractory:integer let
    enable (mode in [OVO, VVI, VVT, VDD, DDD]);
    range 195..430 "El valor debe estar entre 195 y 430.";
    check (self < period) "El refractario ventricular debe ser menor que el
        período";
    check (self < trigger_period) "El refractario ventricular debe ser
        menor que el período de trigger";
    check (self + pace_av < upper_period) "La suma del refractario
        ventricular más el período AV de estimulación debe ser menor que el per
        íodo correspondiente a la frecuencia máxima";
    check (self + sense_av < upper_period) "La suma del refractario
        ventricular más el período AV de sensado debe ser menor que el período
         correspondiente a la frecuencia máxima";
  end
end
```

B.3 Formulario del Censo de 2011

En este anexo se adjunta otro caso de estudio analizado sobre el final del proyecto y presentado en la presentación del proyecto el día 12 de abril de este año. El mismo trata de la implementación de un formulario del censo de Uruguay del año 2011[14].

Contexto

El censo del año 2011 realizado por INE, es el último censo realizado en Uruguay. Los censistas tenían 4 tipos de formularios a completar:

- **Censo de locales:** empresa, si se encuentra abierto, etc.
- **Censo de vivienda:** características de las viviendas desde el exterior, servicios conectados, ocupación, etc.
- **Censo de hogares:** características de las viviendas desde el interior, habitaciones, tenencia, confort, etc.
- **Censo de personas:** parentesco entre las personas del hogar, residencia, situación conyugal, educación, actividades laborales, etc.

Debido a que los formularios son muy extensos por la cantidad de preguntas que tienen, vamos a implementar en InterfLan el formulario de Censo de personas.

Todos los formularios y éste en particular, tiene muchas restricciones de habilitación. Igual muchas de ellas, tienen condiciones encadenadas según las respuestas ingresadas anteriormente.

Implementación del formulario

```
form "Censo 2011 INE - Personas" as censo let
  enum sexo [("Masculino", M), ("Femenino", F)];
  enum si_no [("Si", SI), ("No", NO)];
  enum parentesco [("Jefe de familia", UNO), ("Esposo/a o compañero/a",
      DOS), ("Hijo de ambos", TRES), ("Hijo del esposo/a o compañero/a",
      CUATRO), ("Yerno/nuera", CINCO), ("Hijo de ambos", SEIS), ("
      Padre/Madre", SIETE), ("Suegro/a", OCHO) ,("Hermano/a", NUEVE)
      ,("Cuñado/a", DIEZ), ("Nieto/a", ONCE), ("Otro pariente", DOCE),
      ("Otro no pariente", TRECE), ("Servicio doméstico", CATORCE), ("
      Miembro de hogas colectivo", QUINCE) ];
```

```
enum etnia [("Afro o Negra", UNO), ("Asiática o Amarilla", DOS), ("
    Blanca", TRES), ("Indígena", CUATRO), ("Ninguna", CINCO)];
enum tipo_union [("Casamiento civil", UNO), ("Unión libre con pareja de
    otro sexo", DOS), ("Unión libre con pareja del mismo sexo", TRES)];
enum situacion_cony [("Separado/a de unión libre", UNO), ("Divorciado/
    a", DOS), ("Casado/a", TRES), ("Viudo/a de casamiento", CUATRO),
     ("Viudo/a de unión libre", CINCO), ("Soltero/a", SEIS)];
enum lugar [("En esta localidad", UNO), ("En otra localidad del mismo
    departamento", DOS), ("En otro departamento", TRES), ("En otro paí
    s", CUATRO)];
enum lugar_sin_aqui [("En otra localidad del mismo departamento", UNO)
    , ("En otro departamento", DOS), ("En otro país", TRES)];
enum tiempo_residencia [("Siempre residió aquí", UNO), ("No siempre
    residió aquí", DOS)];
enum preescolar [("Sí, asiste a un centro público", UNO), ("Sí, asiste a
    un CAIF", DOS), ("Sí, asiste a un centro privado", TRES), ("No asiste
    ", CUATRO)];
enum escolar [("Sí, asiste a un establecimiento público", UNO), ("Sí,
    asiste a un establecimiento privado", DOS), ("No asiste pero asisti ó",
     TRES), ("Nunca asistió", CUATRO)];
enum nivel_academico [("Preescolar", UNO), ("Primaria común", DOS),
    ("Primaria especial", TRES), ("Ciclo Básico Liceo (1ero a 3ero)",
    CUATRO), ("Ciclo Básico Liceo UTU (1ero a 3ero)", CINCO), ("
    Bachillerato Secundario (4to a 6to)", SEIS), ("Bachillerato UTU (4to a
    6to)", SIETE), ("Enseñanza Técnica/ Formación Profesional UTU",
    OCHO), ("Magisterio o Profesorado", NUEVE), ("Terciario no
    universitario", DIEZ), ("Carrera de grado o Licenciatura", ONCE), ("
    Postgrado (Diploma/ Maestría/ Doctorado)", DOCE)];
enum requisito_academico [("Maestría completa", UNO), ("Licenciatura/
    Grado universitario completo", DOS), ("Magisterio/ Profesorado
    completo", TRES), ("Bachillerato completo (6to año de Secundaria o
    UTU)", CUATRO), ("Cuarto año de Secundaria completo", CINCO), ("
    Ciclo Básico completo (3er año de Liceo o UTU)", SEIS), ("Primaria
    completa", SIETE), ("Ninguna", OCHO)];
enum tipo_trabajo [("Asalariado/a privado/a", UNO), ("Asalariado/a pú
    blico/a", DOS), ("Miembro de cooperativa de producción", TRES), ("
    Patrón/a", CUATRO), ("Trabajador por cuenta propia", CINCO), ("
    Trabajador familiar no remunerado", SEIS), ("Trabajador de un
    programa social de empleo", SIETE)];
enum nivel_dificultad [("No tiene dificultad", UNO), ("Sí, alguna
    dificultad", DOS), ("Sí, mucha dificultad", TRES), ("Sí, no puede
    hacerlo", CUATRO)];
```

```
enum paises [("Argentina", AR), ("Bolivia", BO), ("Brasil", BR), ("
    Chile", CL), ("Colombia", CO), ("Ecuador", EC), ("Paraguay", PY)
    , ("Perú", PE), ("Venezuela", VE), ("...", OTRO)];
enum departamentos [("Artigas", G), ("Canelones", A), ("Cerro Largo",
     E), ("Colonia", L), ("Durazno", Q), ("Flores", N), ("Florida", O
    ), ("Lavalleja", P), ("Maldonado", B), ("Montevideo", S), ("
    Paysandú", I), ("Rio Negro", J), ("Rivera", F), ("Rocha", C), ("
    Salto", H), ("San José", M), ("Soriano", K), ("Tacuarembó", R), (
    "Treinta y Tres", D)];
enum localidades let
  domain (case dep_censo of
    departamentos.G -> [("Artigas", GA), ("Bella Unión", GBU), ("
        Diego Lamas", GDL)];
    departamentos.A -> [("Canelones", AC), ("Ciudad de la Costa", ACC
        ), ("Las Piedras", ALP), ("La Paz", ALPZ)];
    departamentos.E -> [("Melo", EM), ("Rio Branco", ERB), ("Aceguá
        ", EAC)];
    departamentos.L -> [("Colonia del Sacramento", LCS), ("Carmelo",
        LCR), ("Juan Lacaze", LJL), ("Rosario", LRS)];
    departamentos.Q -> [("Durazno", QD), ("Sarandí del Yí", QSY), ("
        Baygorria", QBY), ("Santa Bernardina", QSB)];
    departamentos.N -> [("Trinidad", NTR), ("Ismael Cortinas", NIC),
        ("Andresito", NAD), ("Cerro Colorado", NCC)];
    departamentos.O -> [("Florida", OFL), ("Sarandí Grande", OSG), ("
        Fray Marcos", OFM), ("Nico Pérez", ONP)];
    departamentos.P -> [("Minas", PM), ("José Pedro Varela", PJPV), (
        "José Batlle y Ordoñez", PJBO), ("Solís de Mataojo", PSM)];
    departamentos.B -> [("Maldonado", BML), ("Punta del Este", BPDE)
        , ("San Carlos", BSC), ("Pan de Azúcar", BPA), ("Piriápolis",
        BPR)];
    departamentos.S -> [("Montevideo", SS)];
    departamentos.I -> [("Paysandú", IP), ("Guichón", IG), ("
        Quebracho", IQ), ("Nuevo Paysandú", INP)];
    departamentos.J -> [("Fray Bentos", JFB), ("Young", JY), ("
        Nuevo Berlín", JNB), ("San Javier", JSJ)];
    departamentos.F -> [("Rivera", FR), ("Tranqueras", FTR), ("Minas
         de Corrales", FMC), ("Vichadero", FVC)];
    departamentos.C -> [("Rocha", CR), ("Castillos", CC), ("Lascano",
         CL), ("Chuy", CCH), ("La Paloma", CLP)];
    departamentos.H -> [("Salto", HS), ("Constitución", HC), ("Belén"
        , HB)];
    departamentos.M -> [("San José de Mayo", MSJ), ("Libertad", ML),
         ("Ciudad del Plata", MCP), ("Ecilda Paullier", MEP), ("Rafael
```

```
        Perazza", MRP)];
    departamentos.K -> [("Mercedes", KM), ("Dolores", KD), ("Villa
        Soriano", KVS), ("Cardona", KC)];
    departamentos.R -> [("Tacuarembó", RT), ("Paso de los Toros", RPT
        ), ("San Gregorio de Polanco", RSGP), ("Villa Ansina", RVA)];
    departamentos.D -> [("Treinta y Tres", DTT), ("Vergara", DV), ("
        Santa Clara del Olimar", DSCO)]
  end);
end


select "Censando en el departamento" as dep_censo:departamentos
    not_empty;
select "Censando en la localidad" as loc_censo:localidades not_empty;


input "Nombre" as nombre:string not_empty;
input "Apellido" as apellido:string not_empty;
select "Sexo" as sexo:sexo not_empty;


input "1. Cuál es su fecha de nacimiento?" as preg_1:datetime
    not_empty;


function calculate_age:integer (bd:datetime) let
  "if(bd){var birthDate = Moment(bd);var dif = Moment().diff(birthDate, '
      years');return dif;}return 0;"
end
label "2. Edad" as preg_2:integer let
  calculate (calculate_age(preg_1));
end


select "3. Qué relación tiene con el jefe del hogar?" as preg_3:
    parentesco not_empty;


select "4. La madre integra este hogar?" as preg_4:si_no let
  enable (preg_3 in [ONCE, DOCE, TRECE, CATORCE] && preg_2 >= 17);
  not_empty (preg_3 in [ONCE, DOCE, TRECE, CATORCE] && preg_2 >=
      17);
end
```

```
select "5. El padre integra este hogar?" as preg_5:si_no let
  enable (preg_3 in [ONCE, DOCE, TRECE, CATORCE] && preg_2 >= 17);
  not_empty (preg_3 in [ONCE, DOCE, TRECE, CATORCE] && preg_2 >=
      17);
end


select "7. Cuál considera su principal ascendencia?" as preg_7:etnia
    not_empty;


select "8. Tiene cónyuge o pareja en el hogar?" as preg_8:si_no let
  enable (preg_2 >= 12);
  not_empty (preg_2 >= 12);
end


input "9. Nombre del cónyuge o pareja" as preg_9:string let
  enable (preg_2 >= 12 && preg_8 in [SI]);
  not_empty (preg_2 >= 12 && preg_8 in [SI]);
end


select "10. Cuál es el tipo de unión?" as preg_10:tipo_union let
  enable (preg_2 >= 12 && preg_8 in [SI]);
  not_empty (preg_2 >= 12 && preg_8 in [SI]);
end


select "11. Actualmente está...? " as preg_11:situacion_cony let
  enable (preg_2 >= 12 && preg_8 != si_no.SI);
  not_empty (preg_2 >= 12 && preg_8 != si_no.SI);
end

select "12. En qué localidad o paraje pasó a residir cuando nació?" as
    preg_12:lugar not_empty;
select "12.1. Nombre de la localidad" as preg_12_1:localidades let
  check (self != loc_censo) "La localidad debe ser distinta de donde se
      está tomando el censo.";
  enable (preg_12 in [DOS]);
  not_empty (preg_12 in [DOS]);
```

```
end
select "12.2. Nombre del departamento" as preg_12_2:departamentos let
  check (self != dep_censo) "El departamento debe ser distinto de donde
      se está tomando el censo.";
  enable (preg_12 in [TRES]);
  not_empty (preg_12 in [TRES]);
end
select "12.3. Nombre del país" as preg_12_3:paises let
  enable (preg_12 == lugar.CUATRO);
  not_empty (preg_12 == lugar.CUATRO);
end


input "13. En qué año aproximadamente llegó a Uruguay para residir en él?"
     as preg_13:integer let
  enable (preg_12 == lugar.CUATRO);
  not_empty (preg_12 == lugar.CUATRO);
end


select "14. Cuánto tiempo hace que reside sin interrupciones en esta
    localidad o paraje?" as preg_14:tiempo_residencia not_empty;
input "14.1. Cuántos años hace que reside aquí?" as preg_14_1:integer
    let
  enable (preg_14 == tiempo_residencia.DOS);
  not_empty (preg_14 == tiempo_residencia.DOS);
end


select "15. Dónde vivía antes de pasar a residir en esta localidad o
    paraje?" as preg_15:lugar_sin_aqui let
  enable (preg_14 == tiempo_residencia.DOS);
  not_empty (preg_14 == tiempo_residencia.DOS);
end
select "15.1. Nombre de la localidad" as preg_15_1:localidades let
  check (self != loc_censo) "La localidad debe ser distinta de donde se
      está tomando el censo.";
  enable (preg_15 in [UNO] && preg_14 == tiempo_residencia.DOS);
  not_empty (preg_15 in [UNO] && preg_14 == tiempo_residencia.DOS)
      ;
end
select "15.2. Nombre del departamento" as preg_15_2:departamentos let
  check (self != dep_censo) "El departamento debe ser distinto de donde
```

```
      se está tomando el censo.";
  enable (preg_15 == lugar_sin_aqui.DOS && preg_14 ==
      tiempo_residencia.DOS);
  not_empty (preg_15 == lugar_sin_aqui.DOS && preg_14 ==
      tiempo_residencia.DOS);
end
select "15.3. Nombre del país" as preg_15_3:paises let
  enable (preg_15 == lugar_sin_aqui.TRES && preg_14 ==
      tiempo_residencia.DOS);
  not_empty (preg_15 == lugar_sin_aqui.TRES && preg_14 ==
      tiempo_residencia.DOS);
end


select "16. En qué localidad o paraje residía en Setiembre de 2006 (hace
    cinco años)?" as preg_16:lugar let
  enable (preg_2 >= 5 && (preg_14 == tiempo_residencia.DOS &&
      preg_14_1 < 5));
  not_empty (preg_2 >= 5 && (preg_14 == tiempo_residencia.DOS &&
      preg_14_1 < 5));
end

select "16.1. Nombre de la localidad" as preg_16_1:localidades let
  check (self != loc_censo) "La localidad debe ser distinta de donde se
      está tomando el censo.";
  enable (preg_16 in [DOS]);
  not_empty (preg_16 in [DOS]);
end
select "16.2. Nombre del departamento" as preg_16_2:departamentos let
  check (self != dep_censo) "El departamento debe ser distinto de donde
      se está tomando el censo.";
  enable (preg_16 == lugar.TRES);
  not_empty (preg_16 == lugar.TRES);
end
select "16.3. Nombre del país" as preg_16_3:paises let
  enable (preg_16 == lugar.CUATRO);
  not_empty (preg_16 == lugar.CUATRO);
end


select "17. Asiste actualmente a un centro de educación inicial o
    preescolar (guardería, jardín de infantes, CAIF, etc.)?" as preg_17:
    preescolar let
```

```
  enable (preg_2 <= 3);
  not_empty (preg_2 <= 3);
end

select "18. Asiste actualmente o asistió alguna vez a un establecimiento
    de enseñanza preescolar, primaria, secundaria, superior o técnica?"
    as preg_18:escolar let
  enable (preg_2 > 3);
  not_empty (preg_2 > 3);
end

select "19. En qué localidad o paraje está ubicado el establecimiento de
    enseñanza al que asiste?" as preg_19:lugar let
  enable (preg_18 in [UNO, DOS]);
  not_empty (preg_18 in [UNO, DOS]);
end
select "19.1. Nombre de la localidad" as preg_19_1:localidades let
  check (self != loc_censo) "La localidad debe ser distinta de donde se
      está tomando el censo.";
  enable (preg_19 in [DOS]);
  not_empty (preg_19 in [DOS]);
end
select "19.2. Nombre del departamento" as preg_19_2:departamentos let
  check (self != dep_censo) "El departamento debe ser distinto de donde
      se está tomando el censo.";
  enable (preg_19 in [TRES]);
  not_empty (preg_19 in [TRES]);
end
select "19.3. Nombre del país" as preg_19_3:paises let
  enable (preg_19 == lugar.CUATRO);
  not_empty (preg_19 == lugar.CUATRO);
end


select "20. Cuál es el nivel que está cursando actualmente?" as preg_20:
    nivel_academico;


select "21. Cuál es el nivel más alto que cursó?" as preg_21:
    nivel_academico not_empty;


select "22. Finalizó ese nivel?" as preg_22:si_no not_empty;
```

```
input "23. Cuántos años aprobó en ese nivel?" as preg_23:integer
    not_empty;


select "24. Para hacer ese curso se exige/exigía:" as preg_24:
    requisito_academico let
  enable (preg_20 in [OCHO, NUEVE, DIEZ, ONCE, DOCE] || preg_21 in
       [OCHO, NUEVE, DIEZ, ONCE, DOCE]);
  not_empty (preg_20 in [OCHO, NUEVE, DIEZ, ONCE, DOCE] || preg_21
       in [OCHO, NUEVE, DIEZ, ONCE, DOCE]);
end


input "25. Cuál es el área, orientación, curso o carrera que estudia/
    estudió? " as preg_25:string let
  enable (preg_20 in [SEIS, SIETE, OCHO, NUEVE, DIEZ, ONCE, DOCE]
      || preg_21 in [SEIS, SIETE, OOCHO, NUEVE, DIEZ, ONCE, DOCE]);
  not_empty (preg_20 in [SEIS, SIETE, OOCHO, NUEVE, DIEZ, ONCE,
      DOCE] || preg_21 in [SEIS, SIETE, OOCHO, NUEVE, DIEZ, ONCE,
      DOCE]);
end


select "26. Sabe leer y escribir ?" as preg_26:si_no let
  enable (preg_2 >= 10 && (preg_18 == escolar.CUATRO || preg_20 in
       [TRES] || preg_21 in [TRES] || ((preg_20 in [DOS] || preg_21
       in [DOS]) && preg_23 <= 3)));
  not_empty (preg_2 >= 10 && (preg_18 == escolar.CUATRO || preg_20
       in [TRES] || preg_21 in [TRES] || ((preg_20 in [DOS] ||
      preg_21 in [DOS]) && preg_23 <= 3)));
end


select "27. Durante la semana pasada, Trabajó por lo menos una hora sin
    considerar los quehaceres de su hogar?" as preg_27:si_no let
  enable (preg_2 >= 12);
  not_empty (preg_2 >= 12);
end


select "28. Hizo algo para afuera o ayudó en un negocio o colaboró en el
```

```
    cuidado de animales, cultivos o huerta que no fuera para su consumo
    propio?" as preg_28:si_no let
  enable (preg_2 >= 12 && preg_27 == si_no.NO);
  not_empty (preg_2 >= 12 && preg_27 == si_no.NO);
end


select "29. Aunque no trabajó la semana pasada, tiene algún trabajo o
    negocio al que seguro volverá?" as preg_29:si_no let
  enable (preg_2 >= 12 && preg_28 == si_no.NO);
  not_empty (preg_2 >= 12 && preg_28 == si_no.NO);
end


select "30. Durante las últimas cuatro semanas, estuvo buscando trabajo o
    tratando de establecer su propio negocio?" as preg_30:si_no let
  enable (preg_2 >= 12 && preg_29 == si_no.NO);
  not_empty (preg_2 >= 12 && preg_29 == si_no.NO);
end

select "31. Ha trabajado antes?" as preg_31:si_no let
  enable (preg_2 >= 12 && preg_30 == si_no.SI);
  not_empty (preg_2 >= 12 && preg_30 == si_no.SI);
end


input "32. Qué tareas realiza (realizaba) en ese trabajo?" as preg_32:
    string let
  enable (preg_27 == si_no.SI || preg_28 == si_no.SI|| preg_29 ==
      si_no.SI || preg_30 == si_no.SI || preg_31 == si_no.SI);
  not_empty (preg_27 == si_no.SI || preg_28 == si_no.SI|| preg_29
      == si_no.SI || preg_30 == si_no.SI || preg_31 == si_no.SI);
end


input "33. Qué produce (producía) o a qué se dedica (dedicaba)
    principalmente la empresa donde trabaja (trabajaba)?" as preg_33:
    string let
  enable (preg_27 == si_no.SI || preg_28 == si_no.SI|| preg_29 ==
      si_no.SI || preg_30 == si_no.SI || preg_31 == si_no.SI);
  not_empty (preg_27 == si_no.SI || preg_28 == si_no.SI|| preg_29
      == si_no.SI || preg_30 == si_no.SI || preg_31 == si_no.SI);
end
```

```
select "34. En ese trabajo es (era):" as preg_34:tipo_trabajo let
  enable (preg_27 == si_no.SI || preg_28 == si_no.SI|| preg_29 ==
      si_no.SI || preg_30 == si_no.SI || preg_31 == si_no.SI);
  not_empty (preg_27 == si_no.SI || preg_28 == si_no.SI|| preg_29
      == si_no.SI || preg_30 == si_no.SI || preg_31 == si_no.SI);
end
```

```
select "37. Es jubilado o pensionista?" as preg_37:si_no let
  enable (preg_2 >= 12);
  not_empty (preg_2 >= 12);
end
```

```
select "38. Es quién realiza los quehaceres del hogar?" as preg_38:
    si_no let
  enable (preg_2 >= 12);
  not_empty (preg_2 >= 12);
end
```

```
input "39. Cuántas hijas e hijos nacidos vivos ha tenido en total?" as
    preg_39:integer let
  enable (sexo == sexo.F && preg_2 >= 12);
  not_empty (sexo == sexo.F && preg_2 >= 12);
end
```

```
input "40. De los hijos e hijas que nacieron vivos, cuántos están vivos
    actualmente?" as preg_40:integer let
  enable (sexo == sexo.F && preg_2 >= 12 && preg_39 != 0);
  not_empty (sexo == sexo.F && preg_2 >= 12 && preg_39 != 0);
end
```

```
input "41. Cuál es la fecha de nacimiento de su último hijo o hija?" as
    preg_41:datetime let
  enable (sexo == sexo.F && preg_2 >= 12 && preg_39 != 0);
  not_empty (sexo == sexo.F && preg_2 >= 12 && preg_39 != 0);
end
```

```
  input "42. En qué año tuvo su primer hijo o hija nacido vivo?" as
      preg_42:integer let
    enable (sexo == sexo.F && preg_2 >= 12 && preg_39 != 0);
    not_empty (sexo == sexo.F && preg_2 >= 12 && preg_39 != 0);
  end

  select "43.1. Tiene dificultad permanente para ver, aún si usa anteojos o
      lentes?" as preg_43_1:nivel_dificultad not_empty;
  select "43.2. Tiene dificultad permanente para oír, aún si usa audífono?"
       as preg_43_2:nivel_dificultad not_empty;
  select "43.3. Tiene dificultad permanente para caminar o subir escalones?"
       as preg_43_3:nivel_dificultad let
    enable (preg_2 >= 2);
    not_empty (preg_2 >= 2);
  end
  select "43.4. Tiene dificultad permanente para entender y/o aprender?" as
       preg_43_4:nivel_dificultad let
    enable (preg_2 >= 6);
    not_empty (preg_2 >= 6);
  end
end
```