Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación
Práctica 5
Semestre 2026-1
22 de Octubre de 2025

Profesora:

- Dra. Karla Ramírez Pulido

Ayudantes:

- Alan Alexis Martínez López

Ayudante de laboratorio:

- Karyme Ivette Azpeitia García

Integrantes del equipo:

- Gonzalez Castillo Patricio Salvador - 321142391
- Valencia Pérez Guillermo Emanuel - 321018689
- Rubio Resendiz Marco Antonio - 320209763
- Sautto Ramirez Seldon - 321084163

1 Objetivos

Analizar y razonar sobre la implementación del intérprete y la gramática de un lenguaje con tipado explícito, así como implementar el verificador de tipos para el lenguaje `Typed-CFWBAEL`. Para llevar a cabo esta tarea, se proporciona la implementación base completa de los archivos `grammars.rkt`, `parser.rkt`, `desugar.rkt`, `interp.rkt`. En estos archivos deberán responder preguntas teóricas y prácticas, también , se debe completar el cuerpo de las funciones faltantes dentro del archivo `verifier.rkt` y contestar los ejercicios siguientes.

La gramática del lenguaje `Typed-CFBWAE` se presenta a continuación:

```
<expr> ::= <id>
         | <num>
         | <bool>
         | {<op> <expr>+}
         | {if <expr> <expr> <expr>}
         | {cond {<expr> <expr>+} {else <expr>}}
         | {with {{<id> : <type> <expr>}+} <expr>}
         | {with* {{<id> : <type> <expr>}+} <expr>}
         | {fun {{<id> : <type>}+} : <type> <expr>}
         | {<expr> {<expr>*}}

<id> ::= a | b | c | ...
<num> ::= 1 | 2 | 3 | ...
<bool> ::= true | false
<op> ::= + | - | * | / | modulo | expt | add1 | sub1
       | < | <= | = | > | >= | not | and | or | zero?

<type> ::= number | boolean | (<type>+ -> <type>)
```

La gramática anterior se define mediante los siguientes tipos en Racket:

```
;; Data-type que define al tipo de dato Type
(define-type Type
  [numberT]
  [booleanT]
  [funT (params (listof Type?))])

;; Definicion del tipo Type-Context
(define-type Type-Context
  [phi]
  [gamma (id symbol?) (tipo Type?) (rest Type-Context?)])

;; Definicion del tipo Binding
(define-type Binding
  [binding (id symbol?) (tipo Type?) (value SCFBWAE?)])

;; Definicion del tipo Param
(define-type Param
  [param (param symbol?) (tipo Type?)])
```

```
;; Definicion del tipo condition para la definicion de cond
(define-type Condition
  [condition (test-expr SCFBWAE?) (then-expr SCFBWAE?)]
  [else-cond (else-expr SCFBWAE?)])

;; Definicion del tipo SCFBWAE
(define-type SCFBWAE
  [idS (i symbol?)]
  [numS (n number?)]
  [boolS (b boolean?)]
  [iFS (condicion SCFBWAE?) (then SCFBWAE?) (else SCFBWAE?)]
  [opS (f procedure?) (args (listof SCFBWAE?))]
  [condS (cases (listof Condition?))]
  [withS (bindings (listof Binding?)) (body SCFBWAE?)]
  [withS* (bindings (listof Binding?)) (body SCFBWAE?)]
  [funS (params (listof Param?)) (rType Type?) (body SCFBWAE?)]
  [appS (fun SCFBWAE?) (args (listof SCFBWAE?))])
```

2 Ejercicios

2.1 Grammar

La sintaxis abstracta y concreta ya se encuentra definida en `grammars.rkt`.

1. **(0.5 pts.)** Explica la diferencia entre `numberT`, `booleanT` y `funT` en la representación de tipos.

   En el lenguaje, el tipo `Type` tiene tres constructores:

   - `numberT` representa el tipo de todos los valores numéricos del lenguaje (expresiones que evalúan a un número).
   - `booleanT` representa el tipo de todos los valores lógicos, es decir, las expresiones que evalúan a `true` o `false`.
   - `funT` representa el tipo de las funciones. En este caso, `funT` guarda una lista de tipos `params` del estilo $[t_1, \ldots, t_n, r]$, donde los primeros elementos corresponden a los tipos de los parámetros y el último elemento r corresponde al tipo de retorno. Así, una anotación gramatical de la forma `(<type>+ -> <type>)` se codifica como un `funT` cuya lista contiene los tipos de todos los argumentos seguidos por el tipo del resultado.

2. **(1 pts.)** ¿Qué ventajas aporta el uso de `type contexts` en un verificador de tipos?

   El `type context` (`Type-Context`) guarda el tipo asociado a cada identificador en un punto del programa. Sus ventajas principales son:

   - **Manejar el alcance:** permite saber qué variables y parámetros están declarados y de qué tipo son en cada expresión (por ejemplo, los que vienen de `with`, `with*` y `fun`).
   - **Detectar errores de tipos:** ayuda a reportar variables no declaradas o cuando se usa una variable con un tipo distinto al que fue declarado.

- **Definir `typeof` de forma clara:** al entrar a un bloque o función se extiende el contexto y al salir se regresa al anterior, haciendo el verificador recursivo y más fácil de implementar.

3. **(0.5 pts.)** ¿Qué papel tienen las construcciones `with` y `with*` en relación con el alcance y el contexto de tipos?

   Las construcciones `with` y `with*` introducen variables locales con tipo, es decir, crean un nuevo alcance y extienden el contexto de tipos con nuevas asociaciones `id : Type`. En `with` las ligaduras se agregan simultáneamente al contexto, mientras que en `with*` se agregan secuencialmente, respetando el orden y permitiendo que unas dependan de otras.

2.2 Verificador

4. **(8 pts.)** Completar el cuerpo de la función `(typeof sexpr context)`, que se encuentra dentro del archivo `verifier.rkt`.

```
;; typeof : SCFBWAE -> Type-Context -> Type
(define (typeof sexpr context) ...)
```

   Dicha función debe tomar una expresión del tipo `SCFBWAE`, verificar que no tenga errores de tipos y devolver el tipo del valor de retorno de cada expresión. En caso de que el programa no cumpla con alguna de las dos condiciones anteriores se debe mandar un error de tipos.

   Operaciones aritméticas y lógicas (`op`): Se debe verificar que el valor de cada parámetro recibido en la lista tenga el tipo correspondiente según el operador. Para los operadores `+`, `-`, `*`, `/`, `=`, `modulo`, `expt`, `add1`, `sub1`, `<`, `<=`, `>`, `>=`, `zero?` los parámetros deben tener tipo `numberT`. Para los operadores `and`, `or`, `not` los parámetros deben tener tipo `booleanT`.

   Condicionales (`if`, `cond`): Se debe verificar que las condicionales en ambos casos sean de tipo `booleanT` o mandar un error de tipos en otro caso. Además, en ambos casos los tipos de las expresiones a evaluar deben ser el mismo para todas las posibles expresiones que se van a ejecutar; es decir, en un `if` la condicional tiene tipo `booleanT` y las expresiones `then-expression` y `else-expression` deben tener el mismo tipo. Análogamente, esto debe suceder con las expresiones de tipo `cond`, donde las `test-expression` tengan tipo `booleanT` y tanto las `then-expression` como las `else-expression` tengan el mismo tipo. Si las expresiones no tienen el mismo tipo se debe mandar un error de tipos.

   Funciones (`fun`): Se debe guardar dentro del contexto el tipo de cada uno de los parámetros recibidos en la función, para que al aplicar la función se puedan obtener los tipos de cada uno de éstos. La función tendrá tipo `funT` $(a_1, a_2, \ldots, r)$, donde cada a_i representa el tipo del parámetro recibido en la posición i, y r es el tipo del valor de retorno de la función.

   Asignaciones locales simples y anidadas (`with`, `with*`): Se debe verificar que los parámetros recibidos se evalúen al mismo tipo al que fueron declarados.

   Aplicaciones de función (`app`): Verificar que la función tenga el tipo `funT` y que los parámetros de la función tengan el tipo correspondiente a la declaración de la función.

2.3 Puntos Extra (Opcionales)

5. **(1 pts.)** Encuentra dos lenguajes que hagan diferente manejo de tipos entre sí en las condicionales `if` y `cond` (o similares), donde expliques si el lenguaje permite diferentes tipos en las múltiples expresiones que se pueden evaluar dependiendo de si se cumple o no la condicional. Adicionalmente, muestra tres instrucciones que ejemplifiquen la explicación anterior. Deberás adjuntar un archivo en formato PDF con este ejercicio.

   Para este ejercicio compararé **Haskell** (lenguaje fuertemente y estáticamente tipado) con **Racket** (lenguaje dinámicamente tipado).

   Haskell:

   En Haskell, las condicionales `if` son expresiones y el sistema de tipos exige que las ramas `then` y `else` tengan el *mismo tipo*. El tipo del `if` es precisamente ese tipo común. Si una rama produce un entero y la otra una cadena, el programa no compila.

```
-- Ejemplo 1 (Haskell, condicional bien tipada):
-- Ambas ramas son Int
if True then 1 else 2      -- Tipo: Int

-- Ejemplo 2 (Haskell, error de tipos):
-- Una rama es Int y la otra es String
if True then 1 else "hola" -- Error de tipos: no compila
```

   En el segundo ejemplo, el compilador rechaza el programa porque no puede asignar un único tipo a la expresión completa `if ... then ... else ...`. Esto muestra que Haskell **no permite** que las distintas ramas de una condicional tengan tipos diferentes.

   Racket:

   En Racket, el tipado es dinámico: las formas `if` y `cond` *no* exigen que todas las ramas tengan el mismo tipo. Cada rama puede devolver valores de tipos distintos y el intérprete simplemente evalúa la rama correspondiente en tiempo de ejecución.

```
;; Ejemplo 3 (Racket, ramas con tipos distintos):
(if #t
    42          ; numero
    "cuarenta") ; cadena

;; Tambien se puede mezclar en un cond:
(cond [(< x 0) "negativo"]   ; cadena
      [else      0])         ; numero
```

   En estos ejemplos de Racket, no hay error de tipos estático aunque una rama regrese un número y otra una cadena. El lenguaje **sí permite** que las distintas expresiones posibles de una condicional (ya sea `if` o `cond`) tengan tipos diferentes; cualquier problema de tipos aparecería, en su caso, hasta tiempo de ejecución.

   En resumen:

   - Haskell requiere que todas las ramas de un `if` (o construcción equivalente) tengan el mismo tipo, y rechaza en compilación programas donde las ramas difieren.

- Racket permite que diferentes ramas de `if` o `cond` regresen valores de tipos distintos, ya que el chequeo de tipos se hace dinámicamente.