Universidad Nacional de Córdoba
Facultad de Astronomía, Matemática y Física

# Compiladores

Descripción del Lenguaje: `COMPI`

2015

---

El proyecto de la materia consiste en implementar un compilador para un lenguaje imperativo simple, similar a `C` o `Pascal`, llamado `COMPI`.

## 1 Consideraciones del Léxico

El lenguaje `COMPI` es *case-sensitive*. Las **palabras reservadas** del lenguaje únicamente estan formadas por minúsculas. Las palabras reservadas y los identificadores son *case-sensitive*. Por ejemplo, **while** es una palabra reservada, pero `WHILE` es un identificador; `cont` and `Cont` son dos nombres diferentes de variables distintas.

Las palabras reservadas son:

**boolean break class continue else false float for if int return true void while extern**

Dos tipos de comentarios son permitidos, aquellos que comienzan con `//` y terminan lal final de la línea (únicamente pueden ser de una línea) y aquellos que estan delimitados por `/*` y `*/` (pueden tener varias líneas de extensión).

Uno o más espacios pueden aparecer entre los símbolos del lenguaje. Llamamos espacios a los espacios en blanco, tabulaciones, saltos de líneas y/o comentarios.

Las palabras reservadas y los identificadores deben estar separados por un espacio o por un símbolo que no es ni una palabra reservada ni un identificador. Por ejemplo, `whiletrue` es un identificador, no dos palabras reservadas.

Los literales del lenguaje son: cadenas de caracteres, enteros y reales. Un literal cadena de caracteres, es una cadena de caracteres (`string`) delimitada por comillas dobles (por ejemplo, `‘‘abc’’`). Una cadena de caracteres esta formada por caracteres ASCII imprimibles, estos literales son iguales a los utilizados en `C`. Por ejemplo, "`\\`" denota el backslash o "`\n`" denota una nueva línea. Los literales enteros y reales son iguales a los utilizados en `C` (por ejemplo 123, 1.23). Los números enteros son de 32 bit con signo, es decir, los valores estan en el rango entre $-2147483648$ y $2147483647$.

# 2 Gramática

**Notación**:

| | |
|---|---|
| ⟨simb⟩ | ⟨simb⟩ es un no-terminal. |
| **simb** | **simb** es un terminal |
| $\left[x\right]$ | cero o una ocurrencia de $x$, *i.e.*, $x$ es opcional; notar que corchetes entre comillas '[' ']' son terminales. |
| $x^*$ | cero o más ocurrencias de $x$. |
| $x^+$**,** | una lista de una o más ocurrencias de $x$'s separadas por coma. |
| $\left\{\ \right\}$ | llaves son usadas para agrupar; notar que llaves entre comillas '{' '}' son terminales. |
| \| | separa alternativas. |

⟨program⟩ → ⟨class_decl⟩$^+$

⟨class_decl⟩ → **class** ⟨id⟩ '{' ⟨field_decl⟩$^*$ ⟨method_decl⟩$^*$ '}'

⟨field_decl⟩ → ⟨type⟩ $\left\{\text{⟨id⟩} \mid \text{⟨id⟩ '[' ⟨int_literal⟩ ']'}\right\}^+$**,** ;

⟨method_decl⟩ → $\left\{\text{⟨type⟩} \mid \textbf{void}\right\}$ ⟨id⟩ ( $\left[\left\{\text{⟨type⟩ ⟨id⟩}\right\}^+\text{,}\right]$ ) ⟨body⟩

⟨body⟩ → ⟨block⟩
| `extern` ;

⟨block⟩ → '{' ⟨field_decl⟩$^*$ ⟨statement⟩$^*$ '}'

⟨type⟩ → **int** | **float** | **boolean** | ⟨id⟩

⟨statement⟩ → ⟨location⟩ ⟨assign_op⟩ ⟨expr⟩ ;
| ⟨method_call⟩ ;
| **if** ( ⟨expr⟩ ) ⟨statement⟩ $\left[\textbf{else}\ \text{⟨statement⟩}\right]$
| **for** ⟨id⟩ = ⟨expr⟩ , ⟨expr⟩ ⟨statement⟩
| **while** ⟨expr⟩ ⟨statement⟩
| **return** $\left[\text{⟨expr⟩}\right]$ ;
| **break** ;
| **continue** ;
| ;
| ⟨block⟩

⟨assign_op⟩ → = | += | -=

⟨method_call⟩ → ⟨id⟩ $\left\{\text{.⟨id⟩}\right\}^*$ ( $\left[\text{⟨expr⟩}^+\text{,}\right]$ )

⟨location⟩ → ⟨id⟩ $\left\{\text{.⟨id⟩}\right\}^*$ | ⟨id⟩ $\left\{\text{.⟨id⟩}\right\}^*$ '[' ⟨expr⟩ ']'

⟨expr⟩ → ⟨location⟩
| ⟨method_call⟩
| ⟨literal⟩
| ⟨expr⟩ ⟨bin_op⟩ ⟨expr⟩
| - ⟨expr⟩
| ! ⟨expr⟩
| ( ⟨expr⟩ )

⟨bin_op⟩ → ⟨arith_op⟩ | ⟨rel_op⟩ | ⟨eq_op⟩ | ⟨cond_op⟩

⟨arith_op⟩ → `+` | `-` | `*` | `/` | `%`

⟨rel_op⟩ → `<` | `>` | `<=` | `>=`

⟨eq_op⟩ → `==` | `!=`

⟨cond_op⟩ → `&&` | `||`

⟨literal⟩ → ⟨int_literal⟩ | ⟨float_literal⟩ | ⟨bool_literal⟩ | ⟨string_literal⟩

⟨id⟩ → ⟨alpha⟩ ⟨alpha_num⟩*

⟨alpha_num⟩ → ⟨alpha⟩ | ⟨digit⟩ | `_`

⟨alpha⟩ → `a` | `b` | `...` | `z` | `A` | `B` | `...` | `Z`

⟨digit⟩ → `0` | `1` | `2` | `...` | `9`

⟨int_literal⟩ → ⟨digit⟩ ⟨digit⟩*

⟨bool_literal⟩ → **true** | **false**

⟨float_literal⟩ → ⟨digit⟩ ⟨digit⟩* . ⟨digit⟩ ⟨digit⟩*

⟨string_literal⟩ → `"` ⟨ASCII⟩* `"`

**Ejemplo de Programa en COMPI**

```
class Program {
    int inc(int x) {
        return x + 1;
    }
    int read_int() extern;
    void print(string s) extern;
    void main() {
        int y;
        y = read_int();
        y = inc(y);
        if (y == 1)
          printf("y==1\n");
        else
          printf("y!=1\n");
    }
}
```

# 3 Semántica

Un programa COMPI consiste de una lista de declaraciones de clases, atributos y métodos. Estos atributos son variables globales que pueden ser accedidas por todos los métodos de la clase y por todos los objetos instanciados de una clase. Los objetos referencian sus atributos e invocan sus métodos con la notacin habitual. Ambos, atributos y mtodos son siempre pblicos ya que el lenguaje no tiene modificadores de visibilidad.

El lenguaje no permite la creación dinámica de objetos.

La declaración de métodos define funciones y procedimientos. El programa debe contener la declaración de una clase llamada **main** con un método llamado del mismo nombre. Este método no tiene parametros. La ejecución de un programa COMPI comienza con el método **main**.

## 3.1 Tipos

Los tipos básicos en COMPI son **int**, **float** y **boolean**.

Se permite la definición de arreglos de enteros, reales y boleanos (`int [`$N$`]`, `float [`$N$`]` y `boolean [`$N$`]`). Solo se permite la definición de arreglos unidimensionales de tamaño fijo (el tamaño es definido en tiempo de compilación). Los arreglos son indexados de 0 a $N-1$, donde $N > 0$ es el tamaño del arreglo. La notación usada para los arreglos es la usual (se utilizan los corchetes).

También se permite la definición de atributos y varibles locales cuyo tipo sea una clase definida por el usuario (objetos).

## 3.2 Reglas de Alcance y Visibilidad de los Identificadores

Las reglas de alcance y visibilidad en COMPI son simples. Primero, todos los identificadores deben ser definidos (textualmente) antes de ser usados. Por ejemplo, una variable debe ser declarada antes de ser usada; un método puede ser invocado únicamente por código ubicado después de su declaración.

En un punto de un programa COMPI existen al menos dos ámbitos (*scopes*) válidos, el global y el local al método. El scope global esta conformado por los identificadores de los atributos y de los métodos declarados al definir el programa. El scope del método esta conformado por los parámetros formales y los identificadores de las variables declaradas en el cuerpo del método. Se pueden definir scope locales adicionales al introducir bloques (⟨block⟩) de código. Los distintos scopes tienen una relación de anidamiento, tal que, el scope global contiene a el scope de los métodos y estos contienen a los scopes de los bloques (los cuales tambien pueden ser declarados de manera anidada). Este anidamiento causa que identificadores definidos en un scope pueda ocultar un identificador con el mismo nombre en scopes superiores. Se debe notar que una variable local puede ocultar tanto un identificador de una variable como de un método.

Los nombres de los identificadores son únicos en cada scope. Es decir, no se puede utilizar el mismo identificador más de una vez en cada ámbito. Por ejemplo, atributos y métodos deben tener distinto nombre en el scope global.

## 3.3 Locaciones en Memoria

El lenguaje COMPI tiene tres clases de locaciones: objetos, variables y arreglos (locales y globales). Cada locación en memoria tiene un tipo. Por ejemplo, las locaciones de tipo **int** y **boolean** contienen valores enteros y lógicos, respectivamente; las locaciones de tipo `int [`$N$`]` denotan arreglos de elementos. Los arreglos son alocados en el espacio de datos estático del programa (frame de ejecución), es decir, no es necesario alocarlos en el heap porque son de tamaño fijo. Los objetos no pueden crearse dinámicamente, por lo cual, también son alocados en el espacio de datos estático del programa (frame de ejecución)

Cada locación es inicializada con un valor por defecto cuando es declarada. Los enteros y reales son inicializados con cero y los boleanos con **false**. Se debe notar que esto implica que cada variable local es inicializada cada vez que se entra al bloque en el que se declara.

## 3.4 Asignaciones

Solo se permiten asignaciones a variables de tipos básicos, es decir, variables de tipos **int**, **float** y **boolean**. La semántica de las asignaciones define la copia del valor. La asignación ⟨location⟩ `=` ⟨expr⟩

copia el valor resultante de evaluar la ⟨expr⟩ en ⟨location⟩ (copia el valor de la expresión en la variable). La asignación ⟨location⟩ `+=` ⟨expr⟩ incrementa el valor almacenado en ⟨location⟩ con ⟨expr⟩. La asignación ⟨location⟩ `-=` ⟨expr⟩ decrementa el valor almacenado en ⟨location⟩ con ⟨expr⟩. Una asignación es válida si ⟨location⟩ y ⟨expr⟩ tienen el mismo tipo. Las asignaciones de incremento y decremento únicamente son permitidas para tipos numéricos.

Un elemento de un arreglo es un elemento de tipo básico, por lo cual, se le pueden asignar valores.

Se permite asignar valores a los parámetros de un método, pero el efecto de estas asignaciones únicamente es visible en el scope del método. Los parámetros son pasados por valor.

### 3.5 Invocación y Retorno de Métodos

La invocación de métodos involucra: (1) pasar los valores de los parámetros reales del método que invoca al invocado; (2) ejecutar el cuerpo del método invocado; y (3) retornar del método invocado, posiblemente retornando un resultado.

Los argumentos son pasados por valor. El valor de evaluar los parámetros reales es copiado a los parámetros formales, los cuales, son considerados como variables locales del método. Los parámetros son evaluados de izquierda a derecha.

Un método solo puede retornar un tipo básico.

Un método que no tiene declarado un tipo de retorno (un método **void**) únicamente puede ser invocado como una sentencia, es decir, no puede ser usado como una expresión. Estos métodos retornan con una sentencia **return** (sin expresión) o cuando el fin del método es alcanzado.

Un método que retorna un resultado puede ser invocado como parte de una expresión. Estos métodos no pueden alcanzar el fin del método, es decir, únicamente retornan con una sentencia **return** (que debe tener asociado una expresión).

Un método que retorna un resultado también puede ser invocado como una sentencia. En este caso, el resultado es ignorado.

### 3.6 Sentencias de Control

**if.** La sentencia **if** tiene la semántica estándar. Primero, la ⟨expr⟩ es evaluada. Si el resultado es **true**, la rama del *then* es ejecutada. En otro caso, se ejecuta la rama del **else**, si existe.

**while.** La sentencia **while** tiene la semántica estándar. Primero, la ⟨expr⟩ es evaluada. Si el resultado es **false**, el cuerpo del ciclo no se ejecuta. En otro caso, el cuerpo del ciclo es ejecutado. Al terminar de ejecutar el cuerpo del ciclo, la sentencia **while** es ejecutada nuevamente.

**for.** En la sentencia **for** el ⟨id⟩ es la variable índice del ciclo (esta es considerada como una declaración local al ciclo) La primer ⟨expr⟩ es el valor inicial de la variable índice del ciclo y la segunda ⟨expr⟩ es el valor final de la variable índice. Estas dos expresiones son evaluadas una sola vez, antes de ejecutar el ciclo por primera vez, y deben ser de tipo entero. El cuerpo del ciclo es ejecutado si el valor corriente de la variable índice es menor al valor final. Después de ejecutar el cuerpo del ciclo el valor del índice es incrementado en 1, y este nuevo valor es comparado con el valor final para poder decidir si se debe ejecutar otra iteración.

**Expresiones.** Las expresiones siguen las reglas usuales de evaluación. En ausencia de otras restricciones, los operadores con la misma precedecia son evaluados de izquierda a derecha. Los paréntesis pueden ser usados para modificar la precedencia usual.

Una locación (variables y elementos de un arreglo) son evaluados al valor que contiene la locación en memoria.

Literales enteros y reales se evalúan a su valor. Literales caracteres se evalúan a su valor ASCII, por ejemplo, `'A'` se evalúa al entero 65.

Los operadores aritméticos (⟨arith_op⟩ y menos unario) y los operadores relacionales (⟨rel_op⟩) tienen el significado y precedencia usual. `%` computa el resto de una división de números enteros.

Los operadores relacionales son usados para comparar expresiones númericas. Los operadores de igualdad (`==`, *es igual*, y `!=`, *no es igual*) son definidos para todos los tipos básicos. únicamente se pueden comparar expresiones del mismo tipo.

El resultado de un operador relacional o de igualdad tienen tipo **boolean**.

Los operadores lógicos `&&` y `||` deben ser evaluados usando evaluación de *corto circuito*. El segundo operador no es evaluado si el primer operador determina el valor de toda la exptresión, es decir, si el resultado es **false** para `&&` o **true** para `||`.

Precedencia de operadores, de mayor precedencia a menor precedencia:

| *Operadores* | *Comentarios* |
|---|---|
| `-` | menos unario |
| `!` | negación lógica |
| `* / %` | multiplicación, división, resto |
| `+ -` | suma, resta |
| `< <= >= >` | relacionales (menor, mayor, ... ) |
| `== !=` | igual y distinto |
| `&&` | conjunción (and) |
| `\|\|` | disyunción (or) |

Notar que estas reglas de precedencia no esta reflejada en la gramática.

**LLamadas a funciones externas.** El lenguaje `COMPI` incluye un mecanísmo, similar al provisto por el lenguaje `C`, para invocar métodos definidos en mdulos externos o bibliotecas.

Un mtodo externo se declara reemplazando su cuerpo por la palabra reservada **extern**.

Esto permite que el compilador pueda realizar chequeo de tipos de manera estndar an en las invocaciones a funciones externas.

Por simplicidad, se recomienda asumir las convenciones de llamadas a funciones C como funciones externas.

## 3.7 Reglas Semánticas

Estas reglas son restricciones (semánticas) adicionales a las reglas sintácticas expresadas en la gramática. Un programa es válido si esta bien formado gramaticálmente y no viola ninguna de las siguientes reglas. El compilador deberá verificar estas reglas, en caso de detectar que no se cumple alguna, deberá generar un mensaje de error que describa el error detectado. Si el compilador no detecta ninguna violación no debera generar ningún informe.

1. Ningún identificador es declarado dos veces en un mismo bloque.
2. Ningún identificador es usado antes de ser declarado.

3. Todo programa contiene la definición de una clase y un método en la misma clase llamado **main**. Este método no tiene parámetros. Notar que la ejecución comienza con el método **main**.
4. El ⟨int_literal⟩ en la declaración de un arreglo debe ser mayor a cero (es la longitud del arreglo).
5. El número y tipos de los argumentos en una invocación a un método debe ser iguales al número y tipos declarados en la definición del método (los parámetros formales y los reales deben ser iguales).
6. Si la invocación a un método es usada como una expresión, el método debe retornar un resultado.
7. Los literales string solo pueden ser usados como argumentos en métodos **externos**.
8. Una sentencia **return** solo tiene asociada una expresión si el método retorna un valor, si le método no retorna un valor (es un método **void**) entonces la sentencia **return** no puede tener asociada ninguna expresión.
9. La expresión en una sentencia **return** debe ser igual al tipo de retorno declarado para el método.
10. Un ⟨id⟩ usado como una ⟨location⟩ debe estar declarado como un parámetro o como una variable local o global.
11. En toda locación de la forma ⟨id⟩ `[`⟨expr⟩`]`
    (a) ⟨id⟩ debe ser una variable arreglo (**array**), y
    (b) el tipo de ⟨expr⟩ debe ser **int**.
12. La ⟨expr⟩ en una sentencia **if** o **while** debe ser **boolean**.
13. Los operandos de ⟨arith_op⟩'s y ⟨rel_op⟩'s deben ser de tipo **int** o **float**.
14. Los operandos de ⟨eq_op⟩'s deben tener el mismo tipo (**int**, **float** o **boolean**).
15. Los operandos de ⟨cond_op⟩'s y el operando de la negación (**!**) deben ser de tipo **boolean**.
16. La ⟨location⟩ y la ⟨expr⟩ en una asignación, ⟨location⟩ `=` ⟨expr⟩, deben tener el mismo tipo.
17. La ⟨location⟩ y la ⟨expr⟩ en una asignación incremental o decremental, ⟨location⟩ `+=` ⟨expr⟩ o ⟨location⟩ `-=` ⟨expr⟩, deben ser de tipo **int** o **float**.
18. Las expresiones (⟨expr⟩) iniciales y finales de un **for** deben ser de tipo **int**.
19. Las sentencias **break** y **continue** solo pueden encontrarse en el cuerpo de un ciclo.

### 3.8 Verificaciones en Tiempo de Ejecución

Todos las verificaciones semánticas descriptas anteriormente son realizadas estáticamente (en tiempo de compilación), pero ciertas verificaciones deben realizarse dinámicamente. Es decir, el generador de código del compilador debe instrumentar código (insertar código) para realizar las verificaciones en tiempo de ejecución.

1. El índice de acceso a una posición de un arreglo debe ser una posición válida (el índice debe estar dentro del rango del arreglo).

Si un error en tiempo de ejecución ocurre debe ser informado y el programa debe terminar. El mensaje de error debe contener suficiente información para detectar cual es el problema en el código fuente.