El manual de referencia de Cool*

Contenido

1 Introducción

Este manual describe el lenguaje de programación Cool: el Lenguaje Orientado a Objetos del Aula. Cool es un lenguaje pequeño que puede implementarse con un esfuerzo razonable en un curso de un semestre. Aun así, Cool conserva muchas de las características de los lenguajes de programación modernos, incluyendo objetos, tipado estático y gestión automática de la memoria.

Los programas Cool son conjuntos de *clases*. Una clase encapsula las variables y procedimientos de un tipo de datos. Las instancias de una clase son *objetos*. En Cool, las clases y los tipos se identifican; es decir, cada clase define un tipo. Las clases permiten a los programadores definir nuevos tipos y procedimientos asociados (o *métodos*) específicos para esos tipos. La herencia permite que los nuevos tipos extiendan el comportamiento de los tipos existentes.

Cool es un lenguaje de *expresión*. La mayoría de las construcciones de Cool son expresiones, y cada expresión tiene un valor y un tipo. Cool es *seguro en* cuanto al tipo: se garantiza que los procedimientos se aplican a datos del tipo correcto. Mientras que la tipificación estática impone una fuerte disciplina en la programación en Cool, garantiza que no pueden surgir errores de tipo en tiempo de ejecución en la ejecución de los programas Cool.

Este manual está dividido en componentes informales y formales. Para una visión general breve e informal, basta con la primera mitad (hasta la sección 9). La descripción formal comienza con la Sección 10.

2 Cómo empezar

El lector que quiera hacerse una idea de Cool desde el principio debería empezar por leer y ejecutar los programas de ejemplo en el directorio ~cs164/examples. Los archivos fuente de Cool tienen la extensión .cl y los archivos de ensamblaje de Cool tienen la extensión .s. El compilador de Cool es ~cs164/bin/coolc. Para compilar un programa:

```
coolc [ -o fileout ] file1.cl file2.cl ... filen.cl
```

El compilador compila los archivos file1.cl hasta filen.cl como si estuvieran concatenados. Cada archivo debe definir un conjunto de clases completas - las definiciones de clase no pueden dividirse en varios archivos. La opción -o especifica un nombre opcional para el código ensamblador de salida. Si no se suministra fileout, el ensamblaje de salida se llama file1.s.

El compilador coolc genera código ensamblador MIPS. Dado que no todas las máquinas que se utilizan en el curso están basadas en MIPS, los programas cool se ejecutan en un simulador de MIPS llamado spim. Para ejecutar un programa cool, escriba

```
% spim
(spim) cargar "archivo.s"
(spim) ejecutar
```

Para ejecutar un programa diferente durante la misma sesión de spim, es necesario reiniciar el estado del simulador antes de cargar el nuevo archivo de montaje:

```
(spim) reinit
```

Una forma alternativa -y más rápida- de invocar spim es con un archivo:

```
spim -archivo.s
```

Este formulario carga el archivo, ejecuta el programa y sale de spim cuando el programa termina.

Asegúrese de que spim es invocado usando el script ~cs164/bin/spim. Puede haber otra versión de spim instalada en algunos sistemas, pero no ejecutará los programas de Cool. Una forma fácil de asegurarse de obtener la versión correcta

es poner el alias de spim en ~cs164/bin/spim. El manual de spim está disponible en la página web del curso y en el lector del curso.

A continuación se muestra una transcripción completa de la compilación y ejecución de ~cs164/examples/list.cl.

Este programa es muy tonto, pero sirve para ilustrar muchas de las características de Cool.

```
% coolc list.cl
% spim -file list.s
SPIM Versión 5.6 del 18 de enero de 1995
Copyright 1990-1994 por James R. Larus (larus@cs.wisc.edu).
Todos los derechos reservados.
Vea en el archivo README un aviso de copyright
completo. Cargado:
/home/ee/cs164/lib/trap.handler
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
Programa COOL ejecutado con éxito
%
```

3 Clases

Todo el código en Cool está organizado en clases. Cada definición de clase debe estar contenida en un único archivo fuente, pero se pueden definir varias clases en el mismo archivo. Las definiciones de clases tienen la forma

```
clase <tipo> [ hereda <tipo> ] {
    <lista_de_características>
};
```

La notación [...] denota una construcción opcional. Todos los nombres de clase son visibles globalmente. Nombres de clases comienzan con una letra mayúscula. Las clases no pueden ser redefinidas.

3.1 Características

El cuerpo de una definición de clase consiste en una lista de definiciones de características. Una característica es un *atributo* o un *método*. Un atributo de la clase A especifica una variable que forma parte del estado de los objetos de la clase A. Un método de la clase A es un procedimiento que puede manipular las variables y los objetos de la clase A.

Uno de los principales temas de los lenguajes de programación modernos es la *ocultación de información*, que es la idea que ciertos aspectos de la implementación de un tipo de datos deben ser abstractos y estar ocultos para los usuarios del tipo de datos. Cool soporta la ocultación de información a través de un mecanismo simple: todos los atributos tienen un alcance local a la clase, y todos los métodos tienen un alcance global. Por lo tanto, la única manera de proporcionar acceso al estado del objeto en Cool es a través de los métodos.

Los nombres de las características deben comenzar con una letra minúscula. Ningún nombre de método puede definirse varias veces en una clase, y ningún nombre de atributo puede definirse varias veces en una clase, pero un método y un atributo pueden tener el mismo nombre.

Un fragmento de `list.cl` ilustra casos sencillos tanto de atributos como de métodos:

```
class Cons hereda de List
{ xcar : Int;
xcdr : Lista;

isNil() : Bool { false };

init(hd : Int, tl : Lista) : Cons {
  {
    xcar <- hd;
    xcdr <- tl;
    self;
  }
}
...
};
```

En este ejemplo, la clase Cons tiene dos atributos xcar y xcdr y dos métodos isNil e init. Nótese que los tipos de los atributos, así como los tipos de los parámetros formales y los tipos de retorno de los métodos, son declarados explícitamente por el programador.

Dado el objeto c de la clase Cons y el objeto 1 de la clase List, podemos establecer los campos xcar y xcdr mediante el método init:

```
c.init(1,l)
```

Esta notación es un *envío orientado a objetos*. Puede haber muchas definiciones de métodos init en muchas clases diferentes. El envío busca la clase del objeto c para decidir qué método init invocar. Como la clase de c es Cons, se invoca el método init de la clase Cons. Dentro de la invocación, las variables xcar y xcdr se refieren a los atributos de c. La variable especial self se refiere al objeto sobre el que se ha enviado el método, que en el ejemplo es el propio c.

Existe una forma especial new C que genera un objeto fresco de la clase C. Un objeto puede pensarse como un registro que tiene una ranura para cada uno de los atributos de la clase, así como punteros a los métodos del
clase. Un envío típico para el método init es:

```
(new Cons).init(1,new Nil)
```

Este ejemplo crea una nueva celda cons y inicializa el "car" de la celda cons para que sea 1 y el "cdr" para que sea nuevo Nil. [1] No existe ningún mecanismo en Cool para que los programadores puedan desasignar objetos. Cool tiene una *gestión automática de la memoria*; los objetos que no pueden ser utilizados por el programa son reasignados por un recolector de basura en tiempo de ejecución.

Los atributos se tratan con más detalle en la sección 5 y los métodos en la sección 6.

3.2 Herencia

Si una definición de clase tiene la forma

```
clase C hereda de P { ... };
```

[1] En este ejemplo, se asume que Nil es un subtipo de List.

entonces la clase C hereda las características de P. En este caso P es la clase *padre* de C y C es una clase *hija* de P.

La semántica de C hereda a P es que C tiene todas las características definidas en P además de sus propias características. En el caso de que un padre y un hijo definan el mismo nombre de método, entonces la definición
dado en la clase hija tiene prioridad. Es ilegal redefinir nombres de atributos. Además, para la seguridad de los tipos, es necesario establecer algunas restricciones sobre cómo se pueden redefinir los métodos (véase la sección 6).

Existe una clase distinguida Objeto. Si la definición de una clase no especifica una clase padre, entonces la clase hereda de Object por defecto. Una clase sólo puede heredar de una única clase, lo que se denomina "herencia única".[2] La relación padre-hijo de las clases define un grafo. Este gráfico no puede contener ciclos. Por ejemplo, si C hereda de P, entonces P no debe heredar de C. Además, si C hereda de
P, entonces P debe tener una definición de clase en alguna parte del programa. Como Cool tiene herencia simple, se deduce que si se cumplen estas dos restricciones, el gráfico de herencia forma un árbol con Object como raíz.

Además de Object, Cool tiene otras cuatro *clases básicas*: Int, String, Bool y IO. Las clases básicas se discuten en la Sección 8.

4 Tipos

En Cool, cada nombre de clase es también un tipo. Además, existe un tipo SELF_TYPE que puede utilizarse en circunstancias especiales.

Una *declaración de tipo* tiene la forma x:C, donde x es una variable y C es un tipo. Toda variable debe tener una declaración de tipo en el momento en que se introduce, ya sea en un let, case o como parámetro formal de un método. También deben declararse los tipos de todos los atributos.

La regla básica de tipos en Cool es que si un método o variable espera un valor de tipo P, entonces se puede utilizar cualquier valor de tipo C en su lugar, siempre que P sea un ancestro de C en la jerarquía de clases. En otras palabras, si C hereda de P, ya sea directa o indirectamente, entonces un C puede ser utilizado donde un P sería suficiente.

Cuando un objeto de la clase C puede utilizarse en lugar de un objeto de la clase P, decimos que C *es conforme* a P o que $C \leq P$ (piénsese que C está más abajo en el árbol de herencia). Como se ha comentado anteriormente, la conformidad se define en términos del grafo de herencia.

Definición 4.1 (Conformidad) Sean A, C y P tipos.

- $A \leq A$ para todos los tipos A
- si C hereda de P, entonces $C \leq P$
- si $A \leq C$ y $C \leq P$ entonces $A \leq P$

Como Object es la raíz de la jerarquía de clases, se deduce que $A \leq$ Object para todos los tipos A.

4.1 SELF_TIPO

El tipo SELF_TYPE se utiliza para referirse al tipo de la variable self. Esto es útil en clases que serán heredadas por otras clases, porque permite al programador evitar especificar un tipo final fijo en el momento de escribir la clase. Por ejemplo, el programa

[2] Algunos lenguajes orientados a objetos permiten que una clase herede de varias clases, lo que se denomina igualmente

"herencia múltiple". "

```
clase Silly {
    copiar() : SELF_TYPE { self };
};

class Sally inherits Silly { }; class

Main {
    x : Sally <- (new Sally).copy();

    main() : Sally { x };
};
```

Dado que se utiliza SELF TYPE en la definición del método copy, sabemos que el resultado de copy es el mismo que el tipo del parámetro self. Así, se deduce que (nueva Sally).copy() tiene el tipo Sally, que se ajusta a la declaración del atributo x.

Tenga en cuenta que el significado de SELF TYPE no es fijo, sino que depende de la clase en la que se utilice. En general, SELF TYPE puede referirse a la clase C en la que aparece, o a cualquier clase que se ajuste a C. Cuando es útil explicitar a qué puede referirse SELF TYPE, utilizamos el nombre de la clase C en la que aparece SELF TYPE como índice SELF_TYPE_C. Esta notación de subíndice no forma parte de la sintaxis de Cool, sino que se utiliza simplemente para aclarar en qué clase aparece una ocurrencia particular de SELF TYPE.

De la definición 4.1 se deduce que $\text{SELF\_TYPE}_X \leq \text{SELF\_TYPE}_X$. También hay una regla de conformidad especial para SELF TYPE:

$$\text{SELF\_TYPE}_C \leq P \text{ si } C \leq P$$

Por último, SELF TYPE puede utilizarse en los siguientes lugares: new SELF TYPE, como tipo de retorno de un método, como tipo declarado de una variable let o como tipo declarado de un atributo. No se permite ningún otro uso de SELF TYPE.

4.2 Tipo Comprobación

El sistema de tipos de Cool garantiza en tiempo de compilación que la ejecución de un programa no puede dar lugar a errores de tipo en tiempo de ejecución. Utilizando las declaraciones de tipo de los identificadores suministrados por el programador, el comprobador de tipos infiere un tipo para cada expresión del programa.

Es importante distinguir entre el tipo asignado por el comprobador de tipos a una expresión en tiempo de compilación, que llamaremos tipo *estático* de la expresión, y el tipo o tipos a los que la expresión puede evaluarse durante la ejecución, que llamaremos tipos *dinámicos*.

La distinción entre tipos estáticos y dinámicos es necesaria porque el verificador de tipos no puede, en tiempo de compilación, tener información perfecta sobre los valores que se calcularán en tiempo de ejecución. Así, en general, los tipos estáticos y dinámicos pueden ser diferentes. Lo que requerimos, sin embargo, es que los tipos estáticos del verificador de tipos sean *sólidos* con respecto a los tipos dinámicos.

Definición 4.2 Para cualquier expresión e, sea $\mathbb{D}_e$ un tipo dinámico de e y sea S_e el tipo estático inferido por el verificador de tipos. Entonces el verificador de tipos es *sólido* si para todas las expresiones e se da el caso de que $\mathbb{D}_e \leq S_e$.

Dicho de otro modo, requerimos que el comprobador de tipos se equivoque al sobrestimar el tipo de una expresión en aquellos casos en los que no es posible una precisión perfecta. Un verificador de tipos así nunca aceptará un programa que contenga errores de tipo. Sin embargo, el precio que se paga es que

el comprobador de tipos rechazará algunos programas que realmente se ejecutarían sin errores en tiempo de ejecución.

5 Atributos

Una definición de atributo tiene la forma

<id> : <tipo> [<- <expr>];

La expresión es una inicialización opcional que se ejecuta cuando se crea un nuevo objeto. El tipo estático de la expresión debe ajustarse al tipo declarado del atributo. Si no se proporciona ninguna inicialización, se utilizará la inicialización por defecto (véase más adelante).

Cuando se crea un nuevo objeto de una clase, deben inicializarse todos los atributos heredados y locales. Los atributos heredados se inicializan primero en orden de herencia, empezando por los atributos de la clase más antigua. Dentro de una clase determinada, los atributos se inicializan en el orden en que aparecen en el texto fuente.

Los atributos son locales a la clase en la que se definen o se heredan. Los atributos heredados no se pueden redefinir.

5.1 Vacío

Todas las variables en Cool se inicializan para contener valores del tipo apropiado. El valor especial void es un miembro de todos los tipos y se utiliza como inicialización por defecto para las variables en las que el usuario no proporciona ninguna inicialización. (void se utiliza donde se usaría NULL en C o null en Java; Cool no tiene nada equivalente al tipo void de C o Java). Tenga en cuenta que no hay ningún nombre para void en Cool; la única forma de crear un valor void es declarar una variable de alguna clase que no sea Int, String o Bool y permitir que se produzca la inicialización por defecto, o almacenar el resultado de un bucle while.

Existe una forma especial isvoid expr que comprueba si un valor es nulo (véase el apartado 7.11). Además, se puede comprobar la igualdad de los valores nulos. Un valor void puede pasarse como argumento, asignarse a una variable o utilizarse de cualquier otra forma en cualquier contexto en el que cualquier valor sea legítimo, excepto que un envío a o un caso sobre void genera un error en tiempo de ejecución.

Las variables de las clases básicas Int, Bool y String se inicializan de forma especial; véase la sección 8.

6 Métodos

La definición de un método tiene la forma

<id>(<id> : <tipo>,...,<id> : <tipo>): <tipo> { <expr> };

Puede haber cero o más parámetros formales. Los identificadores utilizados en la lista de parámetros formales deben ser distintos. El tipo del cuerpo del método debe ajustarse al tipo de retorno declarado. Cuando se invoca un método, los parámetros formales se unen a los argumentos reales y se evalúa la expresión; el valor resultante es el significado de la invocación del método. Un parámetro formal oculta cualquier definición de un atributo del mismo nombre.

Para garantizar la seguridad de tipo, existen restricciones a la redefinición de los métodos heredados. La regla es sencilla: Si una clase C hereda un método f de una clase antecesora P, entonces C puede anular la definición heredada de f siempre que el número de argumentos, los tipos de los parámetros formales y el tipo de retorno sean exactamente los mismos en ambas definiciones.

Para ver por qué es necesaria alguna restricción en la redefinición de los métodos heredados, considere el siguiente ejemplo:

```
clase P {
    f(): Int { 1 };
};

La clase C hereda de P
    { f(): Cadena {
    "1" };
};
```

Sea p un objeto de tipo dinámico P. Entonces

```
p.f() + 1
```

es una expresión bien formada con valor 2. Sin embargo, no podemos sustituir p por un valor de tipo C, ya que el resultado sería añadir una cadena a un número. Por lo tanto, si los métodos pueden redefinirse arbitrariamente, entonces las subclases no pueden simplemente extender el comportamiento de sus padres, y gran parte de la utilidad de la herencia, así como la seguridad de tipos, se pierde.

7 Expresiones

Las expresiones son la mayor categoría sintáctica de Cool.

7.1 Constantes

Las expresiones más sencillas son las constantes. Las constantes booleanas son verdadero y falso. Las constantes enteras son cadenas de dígitos sin signo, como 0, 123 y 007. Las constantes de cadena son secuencias de caracteres encerradas entre comillas dobles, como "Esto es una cadena". Las constantes de cadena pueden tener como máximo 1024 caracteres. Existen otras restricciones para las cadenas; véase la sección 10.

Las constantes pertenecen a las clases básicas Bool, Int y String. El valor de una constante es un objeto de la clase básica correspondiente.

7.2 Identificadores

Los nombres de las variables locales, los parámetros formales de los métodos, self y los atributos de la clase son expresiones. El identificador self puede ser referenciado, pero es un error asignar a self o vincular a self en un let, un case, o como un parámetro formal. También es ilegal tener atributos llamados self.

Las variables locales y los parámetros formales tienen alcance léxico. Los atributos son visibles en toda la clase en

que se declaran o heredan, aunque pueden estar ocultos por declaraciones locales dentro de las expresiones. El enlace de una referencia a un identificador es el ámbito más interno que contiene una declaración para ese identificador, o al atributo del mismo nombre si no hay otra declaración. La excepción a esta regla es el identificador self, que está vinculado implícitamente en todas las clases.

7.3 Asignación

Una asignación tiene la forma

```
<id> <- <expr>
```

El tipo estático de la expresión debe ajustarse al tipo declarado del identificador. El valor es el valor de la expresión. El tipo estático de una asignación es el tipo estático de <expr>.

7.4 Despacho

Existen tres formas de envío (es decir, de llamada a un método) en Cool. Las tres formas difieren únicamente en la forma de seleccionar el método llamado. La forma de envío más utilizada es

<expr>.<id>(<expr>,...,<expr>)

Consideremos el envío $e_0.f(e_1, \ldots, e_n)$. Para evaluar esta expresión, los argumentos se evalúan en orden de izquierda a derecha, desde e_1 hasta e_n. A continuación, se evalúa e_0 y se anota su clase C (si e_0 es nulo se genera un error de ejecución). Por último, se invoca el método f de la clase C, con el valor de e_0 ligado a self en el cuerpo de f y los argumentos reales ligados a los formales como es habitual. El valor de la expresión es el valor devuelto por la invocación del método.

La comprobación de tipo de un envío implica varios pasos. Supongamos que e_0 tiene el tipo estático A. (Recordemos que este tipo no es necesariamente el mismo que el tipo C anterior. A es el tipo inferido por el verificador de tipos; C es la clase del objeto calculado en tiempo de ejecución, que es potencialmente cualquier subclase de A). La clase A debe tener un método f, el envío y la definición de f deben tener el mismo número de argumentos, y el tipo estático del i-ésimo parámetro real debe ajustarse al tipo declarado del i-ésimo parámetro formal.

Si f tiene el tipo de retorno B y B es un nombre de clase, entonces el tipo estático del envío es B. En caso contrario, si f tiene el tipo de retorno SELF TYPE, entonces el tipo estático del envío es A. Para ver por qué esto es sólido, observe que el parámetro self del método f se ajusta al tipo A. Por lo tanto, como f devuelve SELF TYPE, podemos inferir que el resultado también debe ajustarse a A. Inferir tipos estáticos precisos para las expresiones de envío es lo que justifica la inclusión de SELF TYPE en el sistema de tipos Cool.

Las otras formas de envío son:

<id>(<expr>,...,<expr>)

<expr>@<tipo>.id(<expr>,...,<expr>)

La primera forma es la abreviatura de self.<id>(<expr>,...,<expr>).

La segunda forma proporciona una manera de acceder a los métodos de las clases padre que han sido ocultados por redefiniciones en las clases hijo. En lugar de utilizar la clase de la expresión más a la izquierda para determinar el método, se utiliza el método de la clase especificada explícitamente. Por ejemplo, e@B.f() invoca el método
f en la clase B sobre el objeto que es el valor de e. Para esta forma de envío, el tipo estático a la izquierda de
"@" debe ajustarse al tipo especificado a la derecha de "@".

7.5 Condicionales

Un condicional tiene la forma

if <expr> then <expr> else <expr> fi

La semántica de los condicionales es estándar. Primero se evalúa el predicado. Si el predicado es verdadero, se evalúa la rama then. Si el predicado es falso, se evalúa la rama else. El valor del condicional es el valor de la rama evaluada.

El predicado debe tener el tipo estático Bool. Las ramas pueden tener cualquier tipo estático. Para especificar el tipo estático de la condicional, definimos una operación $\sqcup$ (se pronuncia "join") sobre los tipos como sigue. Sean A,B,D cualquier tipo que no sea SELF TYPE. El *menor tipo* de un conjunto de tipos significa el menor elemento con respecto a la relación de conformidad ≤.

$$A \sqcup B = \text{el menor tipo C tal que } A \le C \text{ y } B \le C$$

$$A \sqcup A = A \quad \text{(idempotente)}$$

$$A \sqcup B = B \sqcup A \quad \text{(conmutativo)}$$

$$\text{AUTO TIPO}_D \sqcup A = D \sqcup A$$

Sean T y F los tipos estáticos de las ramas del condicional. Entonces el tipo estático de la condicional es T *LJ* F. (piensa: Camina hacia Object desde cada una de T y F hasta que los caminos se encuentren).

7.6 Bucles

Un bucle tiene la forma

while <expr> loop <expr> pool

El predicado se evalúa antes de cada iteración del bucle. Si el predicado es falso, el bucle termina y se devuelve void. Si el predicado es verdadero, se evalúa el cuerpo del bucle y se repite el proceso. El predicado debe tener el tipo estático Bool. El cuerpo puede tener cualquier tipo estático. El tipo estático de una expresión de bucle es Object.

7.7 Bloques

Un bloque tiene la forma

{ <expr>; ... <expr>; }

Las expresiones se evalúan en orden de izquierda a derecha. Cada bloque tiene al menos una expresión; el valor de un bloque es el valor de la última expresión. Las expresiones de un bloque pueden tener cualquier tipo estático. El tipo estático de un bloque es el tipo estático de la última expresión.

Una fuente ocasional de confusión en Cool es el uso de puntos y comas ("; "). Los puntos y comas se utilizan como terminadores en listas de expresiones (por ejemplo, la sintaxis de bloque anterior) y no como separadores de expresiones. Los semicolones también terminan otras construcciones de Cool, véase la Sección 11 para más detalles.

7.8 Dejemos que

Una expresión let tiene la forma

let <id1> : <tipo1> [<- <expr1>], ..., <idn> : <tipo> [<- <exprn>] in <expr>

Las expresiones opcionales son la *inicialización*; la otra expresión es el *cuerpo*. Un let se evalúa de la siguiente manera. Primero se evalúa <expr1> y el resultado se vincula a <id1>. Luego se evalúa <expr2> y el resultado se vincula a <id2> , y así sucesivamente, hasta que se inicialicen todas las variables de la let. (Si se omite la inicialización de <idk>, se utiliza la inicialización por defecto del tipo <typek>). A continuación se evalúa el cuerpo de la let. El valor de la let es el valor del cuerpo.

Los identificadores let <id1>,...,<idn> son visibles en el cuerpo del let. Además, los identificadores
<id1>,...,<idk> son visibles en la inicialización de <idm> para cualquier m > k.

Si un identificador se define varias veces en un let, las vinculaciones posteriores ocultan las anteriores. Los identificadores introducidos por let también ocultan cualquier definición de los mismos nombres en ámbitos que los contengan. Cada expresión let debe introducir al menos un identificador.

El tipo de una expresión de inicialización debe ajustarse al tipo declarado del identificador. El tipo de let es el tipo del cuerpo.

La <expr> de un let se extiende tan lejos (abarca tantos tokens) como la gramática lo permita.

7.9 Caso

Una expresión de caso tiene la forma

caso <expr0> de
    <id1> : <tipo1> => <expr1>;
    . . .
    <idn> : <typen> => <exprn>;
esac

Las expresiones de caso proporcionan pruebas de tipo en tiempo de ejecución sobre los objetos. Primero, se evalúa expr0 y se anota su tipo dinámico C (si expr0 se evalúa como void se produce un error en tiempo de ejecución). A continuación, de entre las ramas se selecciona la rama con el menor tipo <typek> tal que C ≤ <typek>. El identificador <idk> se vincula al valor de <expr0> y se evalúa la expresión <exprk>. El resultado del caso es el valor
de <exprk>. Si no se puede seleccionar ninguna rama para la evaluación, se genera un error en tiempo de ejecución. Cada caso
debe tener al menos una rama.

Para cada rama, dejemos que T_i sea el tipo estático de <expri>. El tipo estático de una expresión de caso es $\bigsqcup_{1 \le i \le n} T_i$.
El identificador id introducido por una rama de un caso oculta cualquier definición de variable o atributo para id visible
en el ámbito que lo contiene.

La expresión case no tiene una construcción especial para una rama "por defecto" o "de otro modo". El mismo efecto se consigue incluyendo una rama

x : Objeto => ...

porque todo tipo es ≤ a Objeto.

La expresión case proporciona a los programadores una forma de insertar comprobaciones de tipo explícitas en tiempo de ejecución en situaciones en las que los tipos estáticos inferidos por el verificador de tipos son demasiado conservadores. Una situación típica es que un programador escribe una expresión e y la comprobación de tipos infiere que e tiene el tipo estático P. Sin embargo, el programador puede saber que, de hecho, el tipo dinámico de e es siempre C para algún C ≤ P. Esta información puede ser capturada usando una expresión case:

caso e de x : C => ...

En la rama la variable x está ligada al valor de e pero tiene el tipo estático más específico C.

7.10 Nuevo

Una nueva expresión tiene la forma

nuevo <tipo>

El valor es un objeto fresco de la clase apropiada. Si el tipo es SELF TYPE, entonces el valor es un objeto fresco de la clase de self en el ámbito actual. El tipo estático es <tipo>.

7.11 Isvoid

La expresión

isvoid expr

evalúa a true **si** expr **es** void **y evalúa a** false **si** expr **no es** void.

7.12 Operaciones aritméticas y de comparación

Cool tiene cuatro operaciones aritméticas binarias: +, -, *, /. La sintaxis es

expr1 <op> expr2

Para evaluar una expresión de este tipo, primero se evalúa expr1 y luego expr2. El resultado de la operación es el resultado de la expresión.

Los tipos estáticos de las dos subexpresiones deben ser Int. El tipo estático de la expresión es Int. Cool sólo tiene división de enteros.

Cool tiene tres operaciones de comparación: <, <=, =. Para < y <= las reglas son exactamente las mismas que para las operaciones aritméticas binarias, excepto que el resultado es un Bool. La comparación = es un caso especial. Si <expr1> o <expr2> tiene el tipo estático Int, Bool o String, entonces el otro debe tener el mismo tipo estático. Cualquier otro tipo, incluyendo SELF TYPE, puede ser comparado libremente. En los objetos no básicos, la igualdad simplemente comprueba la igualdad de los punteros (es decir, si las direcciones de memoria de los objetos son las mismas). La igualdad se define para void.

En principio, no hay nada malo en permitir pruebas de igualdad entre, por ejemplo, Bool e Int. Sin embargo, una prueba de este tipo debe ser siempre falsa y casi seguro que indica algún tipo de error de programación. Las reglas de comprobación de tipos de Cool capturan estos errores en tiempo de compilación en lugar de esperar hasta el tiempo de ejecución.

Por último, hay un operador aritmético y uno lógico unario. La expresión ~<expr> es el complemento entero de <expr>. La expresión <expr> debe tener el tipo estático Int y toda la expresión tiene el tipo estático Int. La expresión no <expr> es el complemento booleano de <expr>. La expresión <expr> debe tener el tipo estático Bool y toda la expresión tiene el tipo estático Bool.

8 Clases básicas

8.1 Objeto

La clase Object es la raíz del gráfico de herencia. Se definen métodos con las siguientes declaraciones:

```
abort() : Objeto
tipo_nombre() : Cadena
copy() : SELF_TYPE
```

El método abortado detiene la ejecución del programa con un mensaje de error. El método type name devuelve una cadena con el nombre de la clase del objeto. El método copy produce una copia *superficial* del objeto. [3]

8.2 IO

La clase IO proporciona los siguientes métodos para realizar operaciones simples de entrada y salida:

```
out_string(x : Cadena) :
SELF_TYPE out_int(x : Int) :
SELF_TYPE in_string() : Cadena
in_int() : Int
```

[3] Una copia superficial de ***a*** copia ***a*** en sí misma, pero no copia recursivamente los objetos a los que apunta ***a***.

Los métodos out_string y out_int imprimen su argumento y devuelven su autoparámetro. El método in_string lee una cadena de la entrada estándar, hasta un carácter de nueva línea, pero sin incluirlo. El método in_int lee un único número entero, que puede ir precedido de un espacio en blanco. Los caracteres que siguen al entero, hasta la siguiente línea nueva incluida, son descartados por in_int.

Una clase puede hacer uso de los métodos de la clase IO heredando de IO. Es un error redefinir la clase IO.

8.3 Int

La clase Int proporciona números enteros. No hay métodos especiales para Int. La inicialización por defecto de las variables de tipo Int es 0 (no void). Es un error heredar o redefinir Int.

8.4 Cadena

La clase String proporciona cadenas. Se definen los siguientes métodos:

```
length() : Int
concat(s : Cadena) : String
substr(i : Int, l : Int) : Cadena
```

El método length devuelve la longitud del parámetro self. El método concat devuelve la cadena formada al concatenar s después de self. El método substr devuelve la subcadena de su parámetro self que comienza en la posición i con la longitud l; las posiciones de la cadena se numeran comenzando por 0. Se genera un error de ejecución si la subcadena especificada está fuera del rango.

La inicialización por defecto de las variables de tipo String es "" (no void). Es un error heredar o redefinir String.

8.5 Bool

La clase Bool proporciona true y false. La inicialización por defecto de las variables de tipo Bool es false
(no void). Es un error heredar o redefinir Bool.

9 Clase principal

Todo programa debe tener una clase Main. Además, la clase Main debe tener un método main que no tome parámetros formales. El método main debe estar definido en la clase Main (no heredado de otra clase). Un programa se ejecuta evaluando (new Main).main().

Las secciones restantes de este manual proporcionan una definición más formal de Cool. Hay cuatro secciones
que abarca la estructura léxica (sección 10), la gramática (sección 11), las reglas de tipo (sección 12) y la semántica operativa (sección 13).

10 Estructura léxica

Las unidades léxicas de Cool son los enteros, los identificadores de tipo, los identificadores de objeto, la notación especial, las cadenas, las palabras clave y los espacios en blanco.

10.1 Números enteros, identificadores y notación especial

Los enteros son cadenas no vacías de dígitos 0-9. Los identificadores son cadenas (que no son palabras clave) formadas por letras, dígitos y el carácter de subrayado. Los identificadores de tipo comienzan con una letra mayúscula; los identificadores de objeto comienzan con una letra minúscula. Hay otros dos identificadores, **self** y **SELF TYPE** que son tratados especialmente por Cool pero no se tratan como palabras clave. Los símbolos sintácticos especiales (por ejemplo, paréntesis, operador de asignación, etc.) se indican en la figura 1.

10.2 Cuerdas

Las cadenas van entre comillas dobles ''...''. Dentro de una cadena, una secuencia "\c" denota el carácter "c", con la excepción de lo siguiente:

- *¥b* retroceso
- *¥t* ficha
- *¥n* nueva línea
- *¥f* formfeed

Un carácter de nueva línea que no se haya escapado no puede aparecer en una cadena:

''Esto
está
bien"
"Esto no está
bien"

Una cadena no puede contener EOF. Una cadena no puede contener el carácter nulo (\0). Cualquier otro carácter puede ser incluido en una cadena. Las cadenas no pueden cruzar los límites del archivo.

10.3 Comentarios

Hay dos formas de comentarios en Cool. Cualquier carácter entre dos guiones "--" y la siguiente línea nueva (o EOF, si no hay siguiente línea nueva) se tratan como comentarios. Los comentarios también pueden escribirse encerrando el texto en *(∗ . . . ∗)*. Esta última forma de comentario puede estar anidada. Los comentarios no pueden cruzar los límites del archivo.

10.4 Palabras clave

Las palabras clave de cool son: **class, else, false, fi, if, in, inherits, isvoid, let, loop, pool, then, while, case, esac, new, of, not, true.** Salvo las constantes **true** y **false**, las palabras clave no distinguen entre mayúsculas y minúsculas. Para ajustarse a las reglas de otros objetos, la primera letra de **true** y **false** debe ser minúscula; las letras que siguen pueden ser mayúsculas o minúsculas.

10.5 Espacio blanco

El espacio en blanco consiste en cualquier secuencia de los caracteres: blank (ascii 32), \n (newline, ascii 10), \f (form feed, ascii 12), \r (carriage return, ascii 13), \t (tab, ascii 9), \v (vertical tab, ascii 11).

program	::=	[[*clase*;]]$^{+}$
a	::=	**class** TYPE [**hereda de** TYPE] { [[*feature*;]]* }
cara	::=	ID([*formal* [[, *formal*]]*]) : TYPE { *expr* }
cterísti	|	ID : TIPO [<- *expr*]
ca de		
la		
clase		
formal	::=	ID : TIPO
expr	::=	ID <- *expr*
	|	*expr*[@TYPE]. ID([*expr* [[, *expr*]]*])
	|	ID([*expr* [[, *expr*]]*])
	|	**si** *expr* **entonces** *expr* **si no** *expr* **fi**
	|	**while** *expr* **loop** *expr* **pool**
	|	{ [[*expr*;]]$^{+}$ }
	|	**let** ID : TYPE [<- *expr*] [[, ID : TYPE [<- *expr*]]]* **in** *expr*
	|	**case** *expr* **of** [[ID : TYPE => *expr*;]]$^{+}$ **esac**
	|	**nuevo** TIPO
	|	**isvoid** *expr*
	|	*expr* + *expr*
	|	*expr* - *expr*
	|	*expr* * *expr*
	|	*expr* / *expr*
	|	~expr
	|	*expr* < *expr*
	|	*expr* < = *expr*
	|	*expr* = *expr*
	|	**no** *expr*
	|	(*expr*)
	|	ID
	|	entero
	|	cadena
	|	**verd**
	|	**ader**
		o
		falso

Figura 1: Sintaxis genial.

11 Sintaxis genial

La figura 1 ofrece una especificación de la sintaxis de Cool. La especificación no está en pura forma Backus-Naur (BNF); por comodidad, también utilizamos alguna notación de expresión regular. En concreto, A^* significa cero o más A's seguidas; A^+ significa una o más A's. Los elementos entre corchetes [. . .] son opcionales. Doble
Los corchetes [[ ]] no forman parte de Cool; se utilizan en la gramática como un metasímbolo para mostrar la asociación de símbolos gramaticales (por ejemplo, $a[[bc]]^+$ significa a seguido de uno o más pares bc).

11.1 Precedencia

La precedencia de las operaciones binarias infijas y unarias prefijas, de mayor a menor, viene dada por la siguiente tabla:

.
@
~
isvoid
* /
+ -
<= < =
no
<-

Todas las operaciones binarias son asociativas a la izquierda, excepto la asignación, que es asociativa a la derecha, y las tres operaciones de comparación, que no se asocian.

12 Reglas del tipo

Esta sección define formalmente las reglas de tipo de Cool. Las reglas de tipo definen el tipo de cada expresión de Cool en un contexto determinado. El contexto es el entorno de *tipos*, que describe el tipo de cada identificador no vinculado que aparece en una expresión. El entorno de tipos se describe en la Sección 12.1. La sección 12.2 da las reglas de tipo.

12.1 Entornos de tipo

En una primera aproximación, la comprobación de tipos en Cool puede considerarse un algoritmo ascendente: el tipo de una expresión e se calcula a partir de los tipos (previamente calculados) de las subexpresiones de e. Por ejemplo, un entero 1 tiene el tipo Int; en este caso no hay subexpresiones. Como otro ejemplo, si e_n tiene el tipo
X, entonces la expresión { e_1 ; . . . ; e_n ; } tiene tipo X.
Una complicación surge en el caso de una expresión v, donde v es un identificador de objeto. No es posible para decir cuál es el tipo de v en un algoritmo estrictamente ascendente; necesitamos conocer el tipo declarado para v
en la expresión mayor. Esta declaración debe existir para cada identificador de objeto en los programas Cool válidos. Para capturar información sobre los tipos de identificadores, utilizamos un entorno *de tipos*. El entorno consta de tres partes: un entorno de método M , un entorno de objeto O, y el nombre de la clase actual en la que aparece la expresión. El entorno del método y el entorno del objeto

son ambas funciones (también llamadas *mapeos*). El entorno del objeto es una función de la forma

$$O(v) = T$$

que asigna el tipo T al identificador de objeto v. El entorno del método es más complejo; es una función de la forma

$$M(C, f) = (T_1, \ldots, T_{n-1}, T_n)$$

donde C es un nombre de clase (un tipo), f es un nombre de método, y $t_1, \ldots, t_n$ son tipos. La tupla de tipos es la *firma* del método. La interpretación de las firmas es que en la clase C el método f tiene parámetros formales de tipos $(t_1, \ldots, t_{n-1})$-en ese orden- y un tipo de retorno t_n.

Se requieren dos mapeos en lugar de uno porque los nombres de los objetos y los nombres de los métodos no se clasifican, es decir,

puede haber un método y un identificador de objeto con el mismo nombre.

El tercer componente del entorno de tipo es el nombre de la clase actual, que es necesario para las reglas de tipo que implican SELF TYPE.

Cada expresión *e* se comprueba en un entorno de tipo; las subexpresiones de *e* pueden ser de tipo comprobado en el mismo entorno o, si *e* introduce un nuevo identificador de objeto, en un entorno modificado.

Por ejemplo, consideremos la expresión

```
let c : Int <- 33 in
  ...
```

La expresión let introduce una nueva variable c de tipo Int. Sea O el componente de objeto del entorno de tipo para el let. Entonces el cuerpo de la expresión let se comprueba en el entorno de tipos del objeto

$$O[Int/c]$$

donde la notación $O[T/c]$ se define como sigue:

$$O[T/c](c) = T$$
$$O[T/c](d) = O(d) \text{ si } d /= c$$

12.2 Reglas de comprobación de tipos

La forma general de una regla de comprobación de tipo es:

$$\frac{\cdot}{O, M, C \vdash e : T}$$

La regla debe ser leída: En el entorno de tipos de los objetos O, los métodos M, y la clase contenedora C, la expresión e tiene el tipo T. Los puntos sobre la barra horizontal representan otras afirmaciones sobre los tipos de las subexpresiones de e. Estas otras afirmaciones son hipótesis de la regla; si las hipótesis son

se satisface, entonces el enunciado debajo de la barra es verdadero. En la conclusión, el "estilo de giro" ("$f-$") separa el contexto (O, M, C) del enunciado ($e : T$).

La regla para los identificadores de objetos es simplemente que si el entorno asigna un identificador de tipo $Id\ T$, entonces

Id tiene el tipo T.

$$\frac{O(Id) = T\ O,}{M, C \vdash Id : T}$$

La regla de asignación a una variable es más compleja:

$$O(Id) = T\ O, M,$$

$$\frac{e_1 : T' \qquad T' \leq T}{O, M, C \vdash Id \leftarrow e_1 : T'} \quad [\text{Var}]$$

[ASIGNAR]

Tenga en cuenta que esta regla de tipo -al igual que otras- utiliza la relación de conformidad ≤ (véase la Sección 3.2). La regla dice que la expresión asignada e_1 debe tener un tipo T' que se ajuste al tipo T del identificador *Id* en el entorno de tipos. El tipo de la expresión completa es T'.

Las reglas de tipo para las constantes son fáciles:

$$\frac{}{O, M, C \vdash true : Bool} \quad \text{[Verdad]}$$

$$\frac{}{O, M, C \vdash false : Bool} \quad \text{[Falso]}$$

$$\frac{i \text{ es una constante entera}}{O, M, C \vdash i : Int} \quad \text{[Int]}$$

$$\frac{s \text{ es una constante de cadena}}{O, M, C \vdash s : Cadena} \quad \text{[Cadena]}$$

Hay dos casos para el nuevo, uno para el nuevo TIPO DE SELECCIÓN y otro para cualquier otra forma:

$$\frac{T' = \begin{cases} \text{SELF\_TYPE}_C & \text{si } T = \text{SELF\_TYPE} \\ T & \text{de lo contrario,} \end{cases}}{O, M, C \vdash nuevo\ T : T'} \quad \text{[Nuevo]}$$

Las expresiones de envío son las más complejas de comprobar.

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \ldots \\ O, M, C \vdash e_n : T_n \\ T_0' = \begin{cases} C & \text{si } T_0 = \text{SELF\_TYPE}_C \\ T_0 & \text{de lo contrario} \end{cases} \\ M(T_0', f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \le T_i' \quad 1 \le i \le n \\ T_{n+1} = \begin{cases} T_0 & \text{si } T_{n+1}' = \text{SELF\_TYPE} \\ T_{n+1}' & \text{en caso contrario} \end{cases} \end{array}}{O, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}} \quad \text{[Despacho]}$$

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \ldots \\ O, M, C \vdash e_n : T_n \quad T_0 \le T \end{array}}{}$$

$$\frac{M(T,f) = (T'_1, \ldots, T'_n, T'_{n+1}) \quad T_i \leq T'_i \ 1 \leq i \leq n \quad T_{n+1} = \begin{cases} T_0 & \text{si } T'_{n+1} = \text{SELF\_TYPE} \\ T_{n+1} & \text{en caso contrario} \end{cases}}{O, M, C \vdash e_0 @ T.f(e_1, \ldots, e_n) : T_{n+1}} \quad \text{[StaticDispatch]}$$

Para comprobar el tipo de un envío, primero hay que comprobar el tipo de cada una de las subexpresiones. El tipo T_0 de e_0 determina qué declaración del método f se utiliza. Los tipos de los argumentos del envío deben ajustarse a los tipos de los argumentos declarados. Tenga en cuenta que el tipo del resultado del envío es el tipo de retorno declarado o T_0 en el caso de que el tipo de retorno declarado sea SELF TYPE. La única diferencia en la comprobación del tipo de un envío estático es que la clase T del método f se da en el envío, y el tipo T_0 debe ajustarse a T.

Las reglas de comprobación de tipo para las expresiones if y $\{-\}$ son sencillas. Véase la sección 7.5 para la definición de la operación LJ.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : Bool \\ O, M, C \vdash e_2 : T_2 \\ O, M, C \vdash e_3 : T_3 \end{array}}{\begin{array}{l} O, M, C \vdash \text{si } e_1 \text{ entonces } e_2 \text{ si no } e_3 \text{ fi} : T_2 \\ LJ\ T_3 \end{array}} \quad \text{[Si]}$$

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ . \\ O, M, C \vdash e_n : T_n \end{array} \quad O, M,}{C \vdash \{ e_1 ; e_2 ; \ldots e_n ; \} : T_n} \quad \text{[Secuencia]}$$

La regla del permiso tiene algunos aspectos interesantes.

$$\frac{\begin{array}{l} T_0' = \begin{cases} \text{TIPO\_PROPIO}_C & \text{si } T_0 = \text{TIPO\_PROPIO} \\ T_0 & \text{de lo contrario} \end{cases} \\ O, M, C \vdash e_1 : T_1 \\ T_1 \leq T_0' \\ O[T_0'/x], M, C \vdash e_2 : T_2 \end{array}}{\begin{array}{l} O, M, C \vdash \text{dejemos que } x : T_0 \leftarrow \\ e_1 \text{ en } e_2 : T_2 \end{array}} \quad \text{[Let-Init]}$$

En primer lugar, la inicialización e_1 se comprueba de tipo en un entorno sin una nueva definición para x. Así, la variable x no puede utilizarse en e_1 a menos que ya tenga una definición en un ámbito externo. En segundo lugar, el cuerpo e_2 se comprueba el tipo en el entorno O ampliado con la tipificación $x : T_0'$. En tercer lugar, hay que tener en cuenta que el tipo de x puede ser de TIPO PROPIO.

$$\frac{\begin{array}{l} T_0' = \begin{cases} \text{TIPO\_PROPIO}_C & \text{si } T_0 = \text{TIPO PROPIO} \\ T_0 & \text{de lo contrario} \end{cases} \\ O[T_0'/x], M, C \vdash e_1 : T_1 \end{array}}{\begin{array}{l} O, M, C \vdash \text{dejemos } x : T_0 \\ \text{en } e_1 : T_1 \end{array}} \quad \text{[Let-No-Init]}$$

La regla para let sin inicialización simplemente omite el requisito de conformidad. Damos reglas de tipo sólo para una let con una sola variable. La tipificación de una let múltiple

dejemos que $x_1 : T_1$ [← e_1], $x_2 : T_2$ [← e2], . . . , $x_n : T_n$

[← e_n] en e se define como la misma que teclea

let $x_1 : T_1$ [← e_1] in (let $x_2 : T_2$ [← e_2], . . , $x_n : T_n$ [← e_n] in e)

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \quad O[T_1/x_1], M, C \vdash e_1 : T_1' \\ \cdots \\ O[T_n/x_n], M, C \vdash e_n : T_n' \end{array}}{O, M, C \vdash \text{caso } e_0 \text{ de } x_1 : T_1 \Rightarrow e_1 ; \ldots x_n : T_n \Rightarrow e_n ; \text{esac} : \bigsqcup_{1 \le i \le n} T_i'} \quad \text{[Caso]}$$

Cada rama de un caso se comprueba en un entorno donde la variable x_i tiene el tipo T_i. El tipo del caso completo es la unión de los tipos de sus ramas. Las variables declaradas en cada rama de un caso deben tener todas tipos distintos.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : Bool \\ O, M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : Object} \quad \text{[Loop]}$$

El predicado de un bucle debe tener el tipo *Bool*; el tipo de todo el bucle es siempre *Object*. Un isvoid tiene el tipo *Bool*:

$$\frac{O, M, C \vdash e_1 : T_1}{O, M, C \vdash \text{isvoid } e_1 : Bool} \quad \text{[Isvoid]}$$

Con la excepción de la regla de igualdad, las reglas de comprobación de tipos para las operaciones lógicas, de comparación y aritméticas primitivas son sencillas.

$$\frac{O, M, C \vdash e_1 : Bool}{O, M, C \vdash \neg e_1 : Bool} \quad \text{[No]}$$

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : Int \\ O, M, C \vdash e_2 : Int \\ op \in \{<, \le\} \end{array}}{O, M, C \vdash e_1 \; op \; e_2 : Bool} \quad \text{[Compare]}$$

$$\frac{O, M, C \vdash e_1 : Int}{O, M, C \vdash \tilde{} e_1 : Int} \quad \text{[Neg]}$$

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : Int \\ O, M, C \vdash e_2 : Int \\ op \in \{*, +, -, /\} \end{array}}{O, M, C \vdash e_1 \; op \; e_2 : Int} \quad \text{[Arith]}$$

El problema de la regla de igualdad es que cualquier tipo puede ser comparado libremente excepto Int, String y Bool, que sólo pueden compararse con objetos del mismo tipo.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1 \in \{Int, String, Bool\} \vee T_2 \in \{Int, String, Bool\} \Rightarrow T_1 = T_2 \end{array}}{O, M, C \vdash e_1 = e_2 : Bool} \quad \text{[Equal]}$$

Los últimos casos son reglas de comprobación de tipos para atributos y métodos. Para una clase *C*, dejemos que el entorno del objeto O_C dé los tipos de todos los atributos de *C* (incluyendo cualquier atributo heredado). Más formalmente, si x es un atributo (heredado o no) de *C*, y la

$O_C(x)$

declaración de x es $x : T$, entonces

$$( ) = \begin{cases} \text{SELF\_TYPE}_C & \text{si } T = \text{SELF TYPE} \\ T & \text{de lo contrario,} \end{cases}$$

El entorno de métodos M es global para todo el programa y define para cada clase C las firmas de todos los métodos de C (incluidos los métodos heredados).

Las dos reglas para la comprobación de tipo de las definiciones de atributos son similares a las reglas para let. La diferencia esencial es que los atributos son visibles dentro de sus expresiones de inicialización. Nótese que self está ligado en la inicialización.

$$\frac{O_C(x) = T_0 \quad O_C[\text{SELF\_TYPE}_C/self], M, C \vdash e_1 : T_1 \quad T_1 \leq T_0}{O_C, M, C \vdash x : T_0 \leftarrow e_1;} \qquad \text{[Attr-Init]}$$

$$\frac{O_C(x) = T}{O_C, M, C \vdash x : T;} \qquad \text{[Attr-No-Init]}$$

La regla para escribir métodos comprueba el cuerpo del método en un entorno en el que O_C está extendido
con enlaces para los parámetros formales y self. El tipo del cuerpo del método debe ajustarse al tipo de retorno declarado.

$$\frac{M(C, f) = (T_1, \ldots, T_n, T_0) \quad O_C[\text{TIPO\_DE\_TIPO}_C/self][T_1/x_1]\ldots[T_n/x_n], M, C \vdash e_0 : T_0' \quad T_0' \leq \begin{cases} \text{SELF\_TYPE}_C & \text{si } T_0 = \text{SELF\_TYPE} \\ T_0 & \text{en caso contrario} \end{cases}}{O_C, M, C \vdash f(x_1 : T_1, \ldots, x_n : T_n) : T_0 \{ e \};} \qquad \text{[Método]}$$

13 Semántica operativa

Esta sección contiene una presentación principalmente formal de la semántica operativa del lenguaje Cool. La semántica operativa define para cada expresión Cool qué valor debe producir en un contexto determinado. El contexto tiene tres componentes: un entorno, un almacén y un objeto propio. Estos componentes se describen en la siguiente sección. La sección 13.2 define la sintaxis utilizada para referirse a los objetos Cool, y la sección
13.3 define la sintaxis utilizada para referirse a las definiciones de clase.

Tenga en cuenta que una semántica formal es sólo una especificación, no describe una implementación. El propósito de presentar la semántica formal es dejar claros todos los detalles del comportamiento de las expresiones Cool. Cómo se implementa este comportamiento es otra cuestión.

13.1 El medio ambiente y la tienda

Antes de poder presentar una semántica para Cool necesitamos una serie de conceptos y una cantidad considerable de notación. Un *entorno* es un mapeo de identificadores de variables a *ubicaciones*. Intuitivamente, un entorno nos dice para un identificador dado la dirección de la ubicación de memoria donde se almacena el valor de ese identificador. Para una expresión dada, el entorno debe asignar una ubicación a todos los identificadores a los que pueda referirse la expresión. Para la expresión, por ejemplo, $a + b$, necesitamos un entorno que asigne a alguna ubicación y b a alguna ubicación. Utilizaremos la siguiente sintaxis para describir entornos, que es muy similar a la sintaxis de suposiciones

de tipo utilizada en la Sección 12.

$$E = [a : l_1 , b : l_2]$$

Este entorno asigna a la ubicación l_1 , y b a la ubicación l_2 .

El segundo componente del contexto para la evaluación de una expresión es el *almacén* (memoria). El almacén asigna ubicaciones a valores, donde los valores en Cool son simplemente objetos. Intuitivamente, un almacén nos dice qué valor se almacena en una ubicación de memoria determinada. Por el momento, supongamos que todos los valores son enteros. Un almacén es similar a un entorno:

$$S = [l_1 \rightarrow 55, l_2 \rightarrow 77]$$

Esta tienda asigna la ubicación l_1 al valor 55 y la ubicación l_2 al valor 77.

Dados un entorno y un almacén, el valor de un identificador se puede encontrar buscando primero la ubicación a la que se asigna el identificador en el entorno y luego buscando la ubicación en el almacén.

$$E(a) = l_1$$
$$S(l_1) = 55$$

Juntos, el entorno y el almacén definen el estado de ejecución en un paso concreto de la evaluación de una expresión Cool. La doble indirección de los identificadores a las ubicaciones a los valores nos permite modelar las variables. Considere lo que sucede si el valor 99 se asigna a la variable a en el entorno y el almacén definidos anteriormente. Asignar a una variable significa cambiar el valor al que se refiere pero no su ubicación.

Para realizar la asignación, buscamos la ubicación de a en el entorno E y luego cambiamos el mapeo de la ubicación obtenida por el nuevo valor, dando un nuevo almacén S'.

$$E(a) = l_1$$
$$S' = S[99/l_1]$$

La sintaxis $S[v/l]$ denota un nuevo almacén que es idéntico al almacén S, excepto que S' asigna la ubicación l al valor v. Para todas las ubicaciones l' donde $l' \neq l$, seguimos teniendo $S'(l') = S(l')$.

El almacén modela el contenido de la memoria del ordenador durante la ejecución del programa. La asignación a un
modifica el almacén.

También hay situaciones en las que se modifica el entorno. Considere el siguiente fragmento de Cool:

```
let c : Int <- 33 in c
```

Al evaluar esta expresión, debemos introducir el nuevo identificador c en el entorno antes de evaluando el cuerpo del `let`. Si el entorno y el estado actuales son E y S, entonces creamos un nuevo entorno E' y un nuevo almacén S' definidos por:

$$l_c = newloc(S)$$
$$E' = E[l_c/c]$$
$$S' = S[33/l_c]$$

El primer paso es asignar una ubicación para la variable c. La ubicación debe ser nueva, lo que significa que el almacén actual no tiene una asignación para ella. La función *newloc*() aplicada a un almacén nos da una ubicación no utilizada en ese almacén. A continuación, creamos un nuevo entorno E', que asigna c a l_c pero que también contiene todos los
de los mapeos de E para identificadores distintos de c. Obsérvese que si c ya tiene un mapeo en E, el nuevo entorno E' oculta este mapeo antiguo. También debemos actualizar el almacén para asignar la nueva ubicación a un valor. En este caso l_c mapea al valor 33, que es el valor inicial de c definido por la expresión let.

El ejemplo de esta subsección simplifica un poco los entornos y almacenes de Cool, porque los simples

enteros no son valores de Cool. Incluso los enteros son objetos de pleno derecho en Cool.

13.2 Sintaxis de los Objetos Cool

Cada valor de Cool es un objeto. Los objetos contienen una lista de atributos con nombre, un poco como los registros en C. Además, cada objeto pertenece a una clase. Utilizamos la siguiente sintaxis para los valores en Cool:

$$v = X(a_1 = l_1\ ,\ a_2 = l_2\ ,\ \ldots\ ,\ a_n = l_n\ )$$

Lea la sintaxis como sigue: El valor v es un miembro de la clase X que contiene los atributos $a_1\ ,\ \ldots\ ,\ a_n$ cuyas ubicaciones son $l_1\ ,\ \ldots\ .\ l_n$. Obsérvese que los atributos tienen una ubicación asociada. Intuitivamente, esto significa que hay un espacio en memoria reservado para cada atributo.

Para los objetos base de Cool (es decir, Ints, Strings y Bools) utilizamos un caso especial de la sintaxis anterior. Los objetos base tienen un nombre de clase, pero sus atributos no son como los de las clases normales, ya que no pueden ser modificados. Por lo tanto, describimos los objetos base utilizando la siguiente sintaxis:

Int(5)
Bool(*true*)
String(4, "*Cool*")

Para los Ints y Bools, el significado es obvio. Las cadenas contienen dos partes, la longitud y la secuencia real de caracteres ASCII.

13.3 Definiciones de clase

En las reglas presentadas en la siguiente sección, necesitamos una forma de referirnos a las definiciones de atributos y métodos de las clases. Supongamos que tenemos la siguiente definición de clase Cool:

```
clase B {
    s : String <- "Hola";
    g (y:String) : Int {
        y.concat(s)
    };
    f (x:Int) : Int {
        x+1
    };
};

la clase A hereda a la B {
    a : Int;
    b : B <- new B;
    f(x:Int) : Int {
        x+a
    };
};
```

A las definiciones de clase se asocian dos mapeos, denominados *clase* e *implementación*. La *clase* se utiliza para obtener los atributos, así como sus tipos e inicializaciones, de una clase concreta:

$$class(A) = (s : Cadena \leftarrow \text{"}Hola\text{"},\ a : Int \leftarrow 0,\ b : B \leftarrow nueva\ B)$$

Observe que la información de la clase A contiene todo lo que ha heredado de la clase B, así como sus propias definiciones. Si B hubiera heredado otros atributos, éstos también aparecerían en la información de
A. Los atributos se enumeran en el orden en que se heredan y luego en el orden de origen: todos los atributos del mayor antepasado se enumeran primero en el orden en que aparecen textualmente, luego los atributos del siguiente mayor antepasado, y así sucesivamente, hasta los atributos definidos en la clase particular. Nos basamos en este orden para describir cómo se inicializan los nuevos objetos.

La forma general de un mapeo de clases es:

$$class(X) = (a_1 : T_1 \leftarrow e_1 , \ldots , a_n : T_n \leftarrow e_n)$$

Tenga en cuenta que cada atributo tiene una expresión de inicialización, incluso si el programa Cool no especifica una para cada atributo. La inicialización por defecto de una variable o atributo es el *valor por defecto* de su tipo. El valor por defecto de Int es 0, el valor por defecto de String es "", el valor por defecto de Bool es false, y el valor por defecto de cualquier otro tipo es void. [4] El valor por defecto del tipo T se escribe D_T .

El mapeo de implementación proporciona información sobre los métodos de una clase. Para el ejemplo anterior,

La implementación de A se define como sigue:

$$implementación(A, f) = (x, x + a)$$
$$implementación(A, g) = (y, y.concat(s))$$ En

general, para una clase X y un método m

$$implementación(X, m) = (x_1 , x_2 , \ldots , x_n , e_{body})$$

especifica que el método m cuando se invoca desde la clase X, tiene parámetros formales $x_1 , \ldots , x_n$, y el cuerpo del método es la expresión e_{body} .

13.4 Normas de funcionamiento

Equipados con entornos, almacenes, objetos y definiciones de clases, ahora podemos atacar la semántica operativa de Cool. La semántica operativa se describe mediante reglas similares a las utilizadas en la comprobación de tipos. La forma general de las reglas es:

$$\frac{:}{so, S, E \vdash e_1 : v, S'}$$

La regla debe leerse como: En el contexto donde *yo* es el objeto *so*, el almacén es S, y el entorno es E, la expresión e_1 se evalúa al objeto v y el nuevo almacén es S' . Los puntos sobre la barra horizontal representan otras afirmaciones sobre la evaluación de las subexpresiones de e_1 .

Además de un entorno y un almacén, el contexto de evaluación contiene un objeto self *so*. El objeto self es sólo el objeto al que se refiere el identificador self si éste aparece en la expresión. No colocamos self en el entorno y en el almacén porque self no es una variable-no se le puede asignar. Observe que las reglas especifican un nuevo almacén después de la evaluación de una expresión. El nuevo almacén contiene todos los cambios en la memoria que resultan como efectos secundarios de la evaluación de la expresión e_1 .

[4] Un pequeño punto: Estamos permitiendo que void sea usado como una expresión aquí. No hay ninguna expresión para void disponible para los programadores de Cool.

El resto de esta sección presenta y discute brevemente cada una de las reglas operativas. No se tratan algunos casos, que se analizan al final de la sección.

$$\frac{\text{así}, S_1, E \vdash e_1 : v_1 \quad , S_2 \quad E(Id) = l_1 \quad S_3 = S_2[v_1 / l_1]}{\text{así}, S_1, E \vdash Id \leftarrow e_1 : v_1, S_3} \qquad \text{[Asignar]}$$

Una asignación evalúa en primer lugar la expresión del lado derecho, produciendo un valor v_1. Este valor se almacena en memoria en la dirección del identificador.

Las reglas para las referencias a identificadores, self y constantes son sencillas:

$$\frac{E(Id) = l \quad S(l) = v \text{ así},}{S, E \vdash Id : v, S} \qquad \text{[Var]}$$

$$\frac{}{so, S, E \vdash \text{self} : so, S \; so,} \qquad \text{[Self]}$$

$$\frac{}{S, E \vdash \text{true} : Bool(true), S} \qquad \text{[Verdadero]}$$

$$\frac{}{entonces, S, E \vdash \text{false} : Bool(false), S} \qquad \text{[Falso]}$$

$$\frac{i \text{ es una constante entera}}{entonces, S, E \vdash i : Int(i), S} \qquad \text{[Int]}$$

$$\frac{s \text{ es una constante de cadena} \quad l = longitud(s)}{so, S, E \vdash s : Cadena(l, s), S} \qquad \text{[Cadena]}$$

Una nueva expresión es más complicada de lo que cabría esperar:

$$\frac{\text{dynamic\_type}(so) = \begin{cases} X & \text{si } T = \text{TIPO SELF y } so = X(\ldots) \\ T & \text{de lo contrario,} \end{cases} \quad class(T_0) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n) \quad l_i = newloc(S_1), \text{ para } i = 1 \ldots n \text{ y cada } l_i \text{ es diistinto} \quad v_1 = T_0(a_1 = l_1, \ldots, a_n = l_n) \quad S_2 = S_1[D_{T_1} / l_1, \ldots, D_{T_{n/m}}] \quad v_1, S_2, [a_1 : l_1, \ldots, a_n : l_n] \vdash \{a_1 \leftarrow e_1; \ldots; a_n \leftarrow e_n;\} : v_2}{, S_3 \; so, S_1, E \vdash \text{new } T : v_1, S_3} \qquad \text{[Nuevo]}$$

Lo complicado en una nueva expresión es inicializar los atributos en el orden correcto. Ten en cuenta también que, durante la inicialización, los atributos se vinculan al valor por defecto de la clase correspondiente.

$$\frac{\begin{array}{l}\textit{así}, S_1 , E \vdash e_1 : v_1 , \\ S_2\ \textit{así}, S_2 , E \vdash e_2 : v_2 \\ , S_3 \\ . \\ \textit{así}, S_n , E \vdash e_n : v_n , S_{n+1} \\ \textit{así}, S_{n+1} , E \vdash e_0 : v_0 , \\ S_{n+2} \\ v_0 = X(a_1 = l_{a\ 1} , \ldots , a_m = l_{a\ m}) \\ \textit{implementación}(X, f) = (x_1 , \ldots , x_n , e_{n+1}) \\ l_{x\ i} = \textit{newloc}(S_{n+2}), \text{ para } i = 1 \ldots n \text{ y cada } l_{x\ i} \text{ es distinto} \\ S_{n+3} = S_{n+2}[v_1 / l_{x\ 1} , \ldots , v_n / l_{x\ n}] \\ v_0 , S_{n+3} , [a_1 : l_{a\ 1} , \ldots , a_m : l_{a\ m} , x_1 : l_{x\ 1} , \ldots , x_n : l_{x\ n}] \vdash e_{n+1} :\end{array}}{\begin{array}{l} v_{n+1} , S_{n+4}\ so, S_1 , E \vdash e_0 . f(e_1 , \ldots , e_n) : v_{n+1} , S_{n+4}\end{array}}$$ [Despacho]

$$\frac{\begin{array}{l}\textit{así}, S_1 , E \vdash e_1 : v_1 , \\ S_2\ \textit{así}, S_2 , E \vdash e_2 : v_2 \\ , S_3 \\ . \\ \textit{así}, S_n , E \vdash e_n : v_n , S_{n+1} \\ \textit{así}, S_{n+1} , E \vdash e_0 : v_0 , \\ S_{n+2} \\ v_0 = X(a_1 = l_{a\ 1} , \ldots , a_m = l_{a\ m}) \\ \textit{implementación}(T, f) = (x_1 , \ldots , x_n , e_{n+1}) \\ l_{x\ i} = \textit{newloc}(S_{n+2}), \text{ para } i = 1 \ldots n \text{ y cada } l_{x\ i} \text{ es distinto} \\ S_{n+3} = S_{n+2}[v_1 / l_{x\ 1} , \ldots , v_n / l_{x\ n}] \\ v_0 , S_{n+3} , [a_1 : l_{a\ 1} , \ldots , a_m : l_{a\ m} , x_1 : l_{x\ 1} , \ldots , x_n : l_{x\ n}] \vdash e_{n+1} :\end{array}}{\begin{array}{l} v_{n+1} , S_{n+4}\ so, S_1 , E \vdash e_0 @\mathrm{T}.f(e_1 , \ldots , e_n) : v_{n+1} , \\ S_{n+4}\end{array}}$$ [StaticDispatch]

Las dos reglas de envío hacen lo que cabría esperar. Los argumentos se evalúan y se guardan. A continuación, se evalúa la expresión situada a la izquierda del ".". En un envío normal, la clase de esta expresión se utiliza para determinar el método a invocar; en caso contrario, la clase se especifica en el propio envío.

$$\frac{\begin{array}{l}\textit{así}, S_1 , E \vdash e_1 : \textit{Bool}(\textit{true}), S_2 \\ \textit{así}, S_2 , E \vdash e_2 : v_2 , S_3\end{array}}{\begin{array}{l}\textit{así}, S_1 , E \vdash \text{si } e_1 \text{ entonces } e_2 \text{ si no } e_3 \text{ fi} : \\ v_2 , S_3\end{array}}$$ [If-True]

$$\frac{\begin{array}{l}\textit{así}, S_1 , E \vdash e_1 : \textit{Bool}(\textit{false}), S_2 \\ \textit{así}, S_2 , E \vdash e_3 : v_3 , S_3\end{array}}{\begin{array}{l}\textit{así}, S_1 , E \vdash \text{si } e_1 \text{ entonces } e_2 \text{ si no } e_3 \text{ fi} : \\ v_3 , S_3\end{array}}$$ [If-False]

No hay sorpresas en las reglas if-then-else. Tenga en cuenta que el valor del predicado es un objeto Bool, no un booleano.

$$\frac{\begin{array}{l} \textit{sí}, S_1 , E \vdash e_1 : v_1 , S_2 \\ \textit{así}, S_2 , E \vdash e_2 : v_2 , \\ S_3 \\ . \\ \textit{así}, S_n , E \vdash e_n : v_n , S_{n+1} \ \textit{así}, \end{array}}{S_1 , E \vdash \{ e_1 ; e_2 ; \ldots ; e_n ; \} : v_n , \\ S_{n+1}} \quad \text{[Secuencia]}$$

Los bloques se evalúan desde la primera expresión hasta la última, en orden. El resultado es el de la última expresión.

$$\frac{\begin{array}{l}\textit{así}, S_1 \text{ , } E \vdash e_1 : v_1 \text{ , } S_2 \\ l_1 = \textit{newloc}(S_2) \\ S_3 = S_2[v_1 / l_1] \\ E' = E[l_1 / Id] \\ \textit{así}, S_3 \text{ , } E' \vdash e_2 : v_2 \text{ , } S_4\end{array}}{\begin{array}{l}\textit{por lo que}, S_1 \text{ , } E \vdash \text{deje } Id : T_1 \leftarrow e_1 \\ \text{en } e_2 : v_2 \text{ , } S_4\end{array}} \quad \text{[Deja]}$$

Una let evalúa cualquier código de inicialización, asigna el resultado a la variable en una ubicación nueva y evalúa el cuerpo de la let. (Si no hay inicialización, la variable se inicializa al valor por defecto de T_1 .) Damos la semántica operativa sólo para el caso de let con una sola variable. La semántica de un let múltiple

dejemos que $x_1 : T_1 \leftarrow e_1$, $x_2 : T_2 \leftarrow e2, \ldots, x_n : T_n \leftarrow e_n$ en e

se define como lo mismo que

let $x_1 : T_1 \leftarrow e_1$ in (let $x_2 : T_2 \leftarrow e_2, \ldots, x_n : T_n \leftarrow e_n$ in e)

$$\frac{\begin{array}{l}\textit{así}, S_1 \text{ , } E \vdash e_0 : v_0 \text{ ,} \\ S_2\ v_0 = X(\ldots) \\ T_i = \text{ancestro más cercano de } X \text{ en } \{T_1, \ldots, T_n\} \\ l_0 = \textit{newloc}(S_2) \\ S_3 = S_2[v_0 / l_0] \\ E' = E[l_0 / Id_i] \\ \textit{así}, S_3 \text{ , } E' \vdash e_i : v_1 \text{ , } S_4\end{array}}{\begin{array}{l}\textit{así}, S_1 \text{ , } E \vdash \text{caso } e_0 \text{ de } Id_1 : T_1 \Rightarrow e_1 \text{ ; . . ; } Id_n : T_n \Rightarrow e_n \text{ ; esac} : v_1 \text{ ,} \\ S_4\end{array}} \quad \text{[Caso]}$$

Tenga en cuenta que la regla del caso requiere que la jerarquía de clases esté disponible de alguna forma en tiempo de ejecución, para que se pueda seleccionar la rama correcta del caso. Por lo demás, esta regla es sencilla.

$$\frac{\begin{array}{l}\textit{así}, S_1 \text{ , } E \vdash e_1 : \textit{Bool}(\textit{true}), \\ S_2\ \textit{así}, S_2 \text{ , } E \vdash e_2 : v_2 \text{ , } S_3 \\ \textit{así}, S_3 \text{ , } E \vdash \text{mientras } e_1 \text{ bucle } e_2 \text{ pool} :\end{array}}{\textit{void}, S_4} \quad \text{[Loop-True]}$$

$$\frac{}{\begin{array}{l}\textit{así}, S_1 \text{ , } E \vdash \text{mientras } e_1 \text{ bucle } e_2 \text{ pool} : \\ \textit{void}, S_4\end{array}} \quad \text{[Loop-False]}$$

$$\frac{\textit{so}, S_1 \text{ , } E \vdash e_1 : \textit{Bool}(\textit{false}), S_2\ \textit{so},}{S_1 \text{ , } E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \textit{void}, S_2}$$

Hay dos reglas para while: una para el caso en que el predicado es verdadero y otra para el caso en que el predicado es falso. Ambos casos son sencillos. Las dos reglas para isvoid también son sencillas:

$$\frac{\textit{así}, S_1 \text{ , } E \vdash e_1 : \textit{vacío}, S_2}{}$$

a

s

$$\frac{\text{í}, S_1 \text{ , } E \text{ f- isvoid } e_1 : \mathit{Bool}(\mathit{true}), S_2}{\text{así}, S_1 \text{ , } E \text{ f- } e_1 : X(\ldots), S_2} \quad \text{[IsVoid-True]}$$

$$\text{así}, S_1 \text{ , } E \text{ f- isvoid } e_1 : \mathit{Bool}(\mathit{false}), S_2 \quad \text{[IsVoid-False]}$$

El resto de las reglas son para las operaciones aritméticas, lógicas y de comparación primitivas, excepto la igualdad. Todas ellas son reglas sencillas.

$$\frac{\text{así}, S_1, E \vdash e_1 : Bool(b),\ S_2\ v_1 = Bool(\neg b)}{\text{así}, S_1, E \vdash \text{no } e_1 : v_1, S_2} \quad [\text{No}]$$

$$\frac{\text{así}, S_1, E \vdash e_1 : Int(i_1), S_2 \quad \text{así}, S_2, E \vdash e_2 : Int(i_2), S_3 \quad op \in \{\leq, <\} \quad = \begin{cases} Bool(true), \text{ si } i_1\ op\ i_2 \\ Bool(false), \text{ en caso contrario} \end{cases}}{\text{así}, S_1, E \vdash e_1\ op\ e_2 : v_1, S_3} \quad [\text{Comp}]$$

$$\frac{\text{así}, S_1, E \vdash e_1 : Int(i_1),\ S_2\ v_1 = Int(-i_1)}{\text{así}, S_1, E \vdash \sim e_1 : v_1, S_2} \quad [\text{Neg}]$$

$$\frac{\text{así}, S_1, E \vdash e_1 : Int(i_1), S_2 \quad \text{así}, S_2, E \vdash e_2 : Int(i_2), S_3 \quad op \in \{*, +, -, /\} \quad v_1 = Int(i_1\ op\ i_2)}{\text{así}, S_1, E \vdash e_1\ op\ e_2 : v_1, S_3} \quad [\text{Arith}]$$

Los Cool Ints son enteros con signo en complemento a dos de 32 bits; las operaciones aritméticas se definen en consecuencia.

La notación y las reglas anteriores no son lo suficientemente potentes como para describir cómo se comprueba la igualdad de los objetos o cómo se gestionan las excepciones en tiempo de ejecución. Para estos casos recurrimos a una descripción en inglés.

En $e_1 = e_2$, primero se evalúa e_1 y luego e_2. Para comparar la igualdad de los dos objetos, primero se comparan sus punteros (direcciones). Si son iguales, los objetos son iguales. El valor void no es igual a ningún objeto excepto a sí mismo. Si los dos objetos son de tipo String, Bool o Int, se comparan sus respectivos contenidos.

Además, las normas de funcionamiento no especifican qué ocurre en caso de error de ejecución.
La ejecución se aborta cuando se produce un error de ejecución. La siguiente lista especifica todos los posibles errores de ejecución.

1. Un envío (estático o dinámico) en vacío.
2. Un caso sobre el vacío.
3. Ejecución de una sentencia case sin una rama coincidente.
4. División por cero.
5. Subcadena fuera de rango.

6. Desbordamiento del montón.

Por último, las reglas anteriores no explican el comportamiento de ejecución de los envíos a métodos primitivos definidos en las clases Object, IO o String. Las descripciones de estos métodos primitivos se dan en las secciones 8.3-8.5.

14 Agradecimientos

Cool se basa en Sather164, que a su vez se basa en el lenguaje Sather. Algunas partes de este documento se han extraído del manual de Sather164; a su vez, algunas partes del manual de Sather164 se basan en la documentación de Sather escrita por Stephen M. Omohundro.

Varias personas han contribuido al diseño e implementación de Cool, entre ellas Manuel F¨ahndrich, David Gay, Douglas Hauge, Megan Jacoby, Tendo Kayiira, Carleton Miyamoto y Michael Stoddart. Joe Darcy actualizó Cool a la versión actual.