Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de Ciencias de la Computación
CC3032 - Construcción de Compiladores

Especificación del lenguaje YAPL

Dentro de la iniciativa académica, Construcción de Compiladores, enfocaremos el esfuerzo de construir un compilador funcional para el lenguaje de programación **YAPL** (*Yet Another Programming Language*), subconjunto del lenguaje de programación COOL (*Classroom Object-Oriented Language*)

Los programas escritos en YAPL constan de un conjunto de *clases*. Una *clase* encapsula variables y procedimientos de un tipo de dato definido. Una *instancia* de una *clase* se denominan objetos. Cada *clase* definida dentro de YAPL define un nuevo tipo de dato y procedimientos asociados a dichos tipos de datos. La herencia simple es permitida dentro de la definición de nuevos tipos para extender el comportamiento de los tipos de datos existentes.

YAPL implementa un sistema de tipos seguro, es decir que todo objeto dentro de YAPL posee un tipo definido y cada procedimiento debe ser llamado con los tipos adecuados dentro de sus argumentos. Esto garantiza la inexistencia de errores de tipo en tiempos de ejecución.

1. Clases

Todo el código en YAPL se encuentra organizado en clases. Dentro de un archivo pueden existir múltiples definiciones de clases. La definición de una clase tiene la forma:

```
class <type> [ inherits <type> ] {
      <feature_list>
};
```

1.1 Características de una clase

El cuerpo de la definición de una clase consiste en una lista de características. Una característica puede ser un *atributo* o un *método*. Un *atributo* de una clase A especifica una variable que es parte del estado de un objeto particular de la clase A. Un *método* de una clase A es un procedimiento que puede manipular las variables y objetos de la clase A. Algunas propiedades:

- Todos los *atributos* dentro de una clase tienen alcance o ámbito local [dentro de la clase] y todos los *métodos* son de acceso global [o público].
- La única forma de acceder al estado de un objeto en YAPL es por medio de los métodos.
- Los nombres de los atributos y métodos comienzan con letras minúsculas.

- No es posible definir dos o más métodos [atributos] con el mismo nombre dentro de una clase, sin embargo, un método y atributo pueden compartir nombre.

```
class Cons inherits List {
xcar : Int;
xcdr : List;

isNil() : Bool { false };

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

Dado un objeto C de la clase Cons y un objeto L de la clase List, se puede inicializar los atributos xcar y xcdr por medio del método init

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

Esta notación se basa en el concepto de polimorfismo en tiempo de compilación [Static Dispatch]. Otra forma de inicializar atributos dentro de objetos es utilizando el operador new:

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

1.2 Herencia

Si la definición de una clase tiene la forma:

```
class C inherits P { ... };
```

Entonces la clase C hereda los atributos de P. En este caso se dice que P es clase padre de C o que C es una clase hija de P. Algunas observaciones:

- Solamente se permite herencia simple.
- La herencia [C inherits P] significa que todos los atributos definidos en P son heredados a la clase C.
- No es posible re-definir nombres de atributos.

- Si existen nombres de métodos similares en la clase hijo que la clase padre, la clase hijo toma precedencia.
- Existe una clase **Object**. Si la definición de una clase no especifica clase padre, entonces la clase hereda de **Object**.
- La relación padre-hijo en clases define un grafo. Este grafo no permite ciclos.
- Existen 4 clases básicas adicionales: **Int, String, Bool y IO**

2. Tipos de datos

En YAPL toda definición de una clase genera un tipo de dato nuevo. Además, existe un tipo especial SELF_TYPE utilizado en ciertas circunstancias.

- Una *declaración* tiene la forma x:C, donde x es una variable y C es un tipo.
- Toda variable debe poseer un tipo de dato.
- Todo atributo debe de declararse con un tipo de dato.
- Si C hereda de P, directa o indirectamente, entonces C puede utilizarse en contextos donde P es utilizado [herencia de tipos]. Esto es C se ajusta a P [C conforms to P], denotado como C <= P
- Sean A, C y P tipos de datos en YAPL, entonces
    - A<=A para todo A
    - Si C hereda de P entonces C<=P
    - Si A<=C y C<=P entonces A<=P

2.1 SELF_TYPE

El tipo de dato SELF_TYPE se usa para representar el tipo de dato de la variable **self** [similar a puntero this en c++].

```
class Silly {
     copy() : SELF_TYPE { self };
};

class Sally inherits Silly { };

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

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

2.2 Comprobación de Tipos

Durante la fase de análisis semántico del lenguaje YAPL, se implementará el sistema de tipos que garantiza que en tiempo de ejecución de un programa no pueden existir errores de tipo. Esto se hará asignando a cada expresión del programa escrito un tipo de datos en tiempo de compilación. Este tipo de dato se denomina *tipo estático* de la expresión.

3. Atributos

La definición de un atributo [variable] tiene la forma:

```
<id> : <type> [ <- <expr> ];
```

- La expresión es una inicialización opcional para el atributo creado.
- El tipo estático de la expresión debe coincidir con el tipo del atributo declarado
- Si no existe expresión para inicializar el atributo, se utiliza la inicialización *default.* [Int -> 0, Bool -> false, String->””]
- Cuando se crea un objeto a partir de una clase, todos los atributos [locales y heredados] deben ser inicializados. El orden de inicialización es acorde a la jerarquía de herencia y al orden en el que aparecen en el programa fuente.
- Los atributos definidos o heredados tienen alcance local dentro de la clase.
- Los atributos heredados no pueden re-definirse.

3.1 void

El valor especial **void** es usado como valor default de inicialización para variables en donde el programador no especifique explícitamente la inicialización. [Similar a null en Java].

- Existe un operador especial **isvoid expr** que valida si la expresión es **void**

4. Métodos [Procedimientos / Funciones]

Un método se define de la siguiente forma:

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

- Pueden existir cero o más parámetros formales.
- Los nombres de los parámetros formales son únicos entre sí.
- El tipo del valor de retorno debe coincidir con la declaración del tipo del método.
- Los parámetros formales ofuscan cualquier definición de atributos con el mismo nombre.
- Si la clase C hereda un método f de la clase ancestra P, entonces C puede sobrecargar la definición heredada de f si y sólo si la firma de la sobrecarga coincide con el método original. [Preserva la seguridad en el sistema de tipos]

5. Expresiones

- Expresiones simples
    - Constantes booleanas: **true, false**
    - Constantes enteras, cadenas de caracteres números con signo y sin signo: 0, 123, -3, 007
    - Constantes tipo cadena, secuencia de caracteres encerrados en doble comillas: **"This is a string."**
- Identificadores
    - Los nombres de variables locales, parámetros formales, atributos de clases son identificadores.
    - No es posible tener un atributo llamado **self** [autoreferencia a la clase]
    - Variables locales y parámetros formales poseen un ámbito léxico [ámbito local y ocultamiento de variables dentro de métodos]
    - El enlace a un identificador se realiza al ámbito léxico más cercano que contiene la declaración para dicho identificador o el atributo con el mismo nombre si no existe dicha declaración.
- Asignaciones
    - Una asignación tiene la forma `<id> <- <expr>`
    - El tipo estático <expr> debe coincidir con el tipo declarado para el <id>
    - El valor de <expr> del lado derecho se convierte en el valor del objeto <id>
    - El tipo de dato de la asignación es el tipo de dato de <expr>
- Llamadas a métodos
    - Existen 3 formas de realizar la llamada a un método:
        - `<expr>.<id>(<expr>,...,<expr>)`
        - `<id>(<expr>,...,<expr>)`
        - `<expr>@<type>.id(<expr>,...,<expr>)`
    - La primera forma evalúa <expr>, obteniendo la clase a la cual hace referencia, C por ejemplo, para luego invocar el método <id> dentro de la clase C, con los parámetros actuales <expr>,<expr>, …<expr>. El objeto **self** hará referencia al objeto <expr> instanciado de la clase C.
    - El tipo estático de la llamada al método es el tipo asignado al tipo del método <id>.
    - El objeto de retorno del método <id> debe coincidir con el tipo del método declarado.
    - El segundo método es una forma corta de escribir `self.<id>(<expr>,...,<expr>)`
    - El tercer método es una forma de llamar a métodos de clases padres <type> que se encuentran ocultas por la redefinición en clases hijas.
- Condicionales
    - Un condicional tiene la forma `if <expr1> then <expr2> else <expr3> fi`
    - `<expr1>` debe ser de tipo estático Bool

- Si el predicado es true, <expr2> es evaluado, de lo contrario <expr3> se evalúa.
- El valor de la expresión if, es el valor de la rama evaluada.
- Sea A el tipo de <expr2> y B el tipo de la <expr3>, entonces el tipo del condicional es el tipo del objeto C tal que A<=C y B<=C

- Ciclos
  - Un ciclo tiene la forma `while <expr1> loop <expr2> pool`
  - `<expr2>` se evalúa mientras `<expr1>` sea true
  - El tipo de <expr1> debe ser **Bool**
  - El tipo del ciclo completo es **Object**
  - El valor del ciclo es **void**.
- Bloques
  - Un bloque en YAPL se define como `{ <expr>; ... <expr>; }`
  - Todo bloque contiene al menos una expresión.
  - Las expresiones son evaluadas de izquierda a derecha.
  - El valor del bloque es el valor de la última expresión evaluada.
  - El tipo del bloque es el tipo de la última expresión evaluada.
- **Let**
  - Una expresión let tiene la forma:
    - `let <id1> : <type1> [ <- <expr1> ], ..., <idn> : <typen> [ <- <exprn> ] in <expr>`
  - Las expresiones opcionales son incializaciones
  - La expresión final es el cuerpo de la función expr.
  - <expr1> se evalúa y se asigna a <id1>, <expr2> se evalúa y se asigna a <id2>, y así sucesivamente, luego se evalúa el cuerpo <expr>
  - Los identificadores <idk> son visibles en el cuerpo de la función **let**. Estos identificadores con tratados como *variables locales*
  - Al menos debe de existir un <id> dentro de la definición del **let**
  - El tipo de una expresión de declaración es el <type> declarado.
  - El tipo de la instrucción <let> es el tipo del respectivo cuerpo <expr>
  - *El cuerpo de la expresión <expr> del cuerpo de **let** se extiende la mayor cantidad de tokens posibles que sean permitidos*.
- Operador **new**
  - `new <type>` devuelve un objeto de la clase apropiada <type>
  - Si <type> es SELF_TYPE, devuelve un objeto de la clase **self**
  - El tipo devuelto es <type>
- Operador **isvoid**
  - `isvoid expr` se evalua a **true** si expr es **void**, **false** de lo contrario.
- Operaciones Aritméticas y de Comparación
  - YAPL contiene cuatro operaciones aritméticas: +, -, *, /
  - YAPL contiene tres operaciones de comparación: <, <=, =
  - YAPL contiene dos operaciones unarias: **~** [complemento a variables tipo **Int**]**, not** [complemento a variables tipo **Bool**]

- La sintaxis de una operación binaria es `expr1 <op> expr2`
- La sintaxis de una operación unaria es `<op> expr`

6. Clases Básicas

6.1 Object

La clase Object es la raíz del grafo de herencia. Esta clase posee métodos con la siguiente definición:

```
abort(): Object
type_name(): String
copy(): SELF_TYPE
```

El método abort() detiene la ejecución del programa con un mensaje de error. El método type_name, retorna una cadena con el nombre de la clase a la que pertenece el objeto. El método copy, crea una copia *shallow* del objeto.

6.2 IO

La clase IO provee métodos simples para operaciones de entrada y salida

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

- Los métodos out_string y out_int, colocan en pantalla el parámetro x.
- El método in_string() lee una cadena de la entrada estándar, hasta un caracter de nueva línea (sin incluirlo).
- El método in_int() lee un número entero de la entrada estándar.
- Es un error definir de nuevo la clase con nombre IO.

6.3 Int

La clase Int, permite la creación de variables tipo entero.

- No existen métodos especiales.
- Inicialización default es 0
- No es posible heredar o redefinir **Int**

6.4 String

La clase String, permite la creación de variables tipo cadena. Esta clase posee métodos con la siguiente definición:

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

- El método length() devuelve la longitud del parámetro **self**
- El método concat, concatena el parámetro **s** a la derecha de **self**
- El método substr, devuelve la subcadena del **self**, comenzando en la posición **i,** con una longitud **l**. La posición de los caracteres en la cadena comienza en 0.
- En tiempo de ejecución, el método substr devuelve un error si los índices están fuera de rango.
- El valor default de inicialización para un **String** es "" [cadena vacía]
- No es posible heredar o redefinir **String**

6.5 Bool

La clase Bool, permite la creación de variables tipo boolean.

- No existen métodos especiales.
- Inicialización default es **false.**
- No es posible heredar o redefinir **Bool**

7. Referencias

YAPL se encuentra basado en el lenguaje COOL y este documento se basa principalmente en el manual de COOL. Las personas involucradas en el desarrollo de COOL son Manuel Fahndrich, David Gay, Douglas Hauge, Megan Jacoby, Tendo Kayiira, Carleton Miyamoto, y Michael Stoddart.