TPE

72.39 - Autómatas, Teoría de Lenguajes y Compiladores

PACO ++

Grupo: Sensibles al Contexto

BALAGUER Pedro - 55795 <*pbalaguer@itba.edu.ar*>
BENÍTEZ Julián - 56283 <*jubenitez@itba.edu.ar*>
GARRIGÓ Mariano - 54393 <*mgarrigo@itba.edu.ar*>
PARODI Eliseo - 56399 <*eparodi@itba.edu.ar*>

Índice

1_ Idea Inicial

La idea inicial del proyecto era que el lenguaje sea no tipado y que todas las operaciones sean soportadas entre cualquier tipo de datos (ej: multiplicar int por string). Además, la sintaxis sería parecida a la de Octave.

El lenguaje es llamado Paco++ (inspirado en el profesor de *Probabilidad y Estadística* y *Métodos Numéricos*) y compila a C.

2_ Consideraciones

Manteniendo la idea central del proyecto, las operaciones suma, resta, multiplicación y división entre los tipos básicos (ver sección siguiente) están definidas y no darán error. Algunos ejemplo son:

- sumar/restar/multiplicar/dividir un entero con un decimal se obtiene un decimal aplicando la operación adecuada.
- multiplicar un entero o un decimal n por un string repite el string n veces, y lo invierte si n es negativo.
- dividir un string por otro (a/b) elimina los substrings b en a.
- multiplicar una lista por otro objeto multiplica cada elemento de la lista por el objeto.

Este comportamiento se puede apreciar correctamente al compilar y ejecutar el código *operations.ppp*.

Todas estas operaciones pueden encontrarse implementadas en *operations.c, sum.c, sub.c, mul.c* y *div.c* para mayor información acerca de ellas.

3_ Desarrollo del TP

Del objetivo inicial mantuvimos como eje central la idea de que las operaciones puedan realizarse entre cualquier tipo. Para ello creamos un struct objeto (*_object* definido en *object.h*) que tiene una referencia a su tipo y un UNION con cada uno de los tipos básicos. Además, creamos una librería de operaciones con todas las operaciones entre tipos definidos que es usada por el archivo generado en C en compilación. Con esto, el código generado en C definirá todas las variables con este tipo y podrá aplicar todas las operaciones posibles (las operaciones reciben todas *_object* y retornan *_object*).

Utilizando como base una implementación propia de hashtable en C, utilizamos una tabla de tipos y otra para las operaciones. La primera se encuentra en *paco.y*

y permite asociar el nombre de una variable con el *struct y_variable* (ver *yaccObjects.h*) que contiene información necesaria para que el compilador ejecute correctamente. La tabla de operaciones se crea con el llamado a la función *createOpTable()* y crea una matriz cúbica con índices x, y, z asociados al tipo de operación (*enum OpValue* en *types.h*), tipo del primer operando (*enum TypeID* en *types.h*) y tipo del segundo operando respectivamente. Al acceder con esas tres claves se almacena un puntero a función para poder ejecutar la función.

4_ Gramática

Los tipos soportados nativamente son :

- Number: entero positivo o negativo (ej: 52)
- Decimal: número decimal con coma (ej: 10.4)
- String: cadena de caracteres (ej: "paco")
- Lista: conjunto de objetos de cualquier tipo (ej: [1, 1.5, "asd", [1, 2,"asd"]]

Las instrucciones deben ocupar una línea y deben estar separadas por '*n*'. Además, inspirándonos en Octave, las instrucciones pueden terminar o no por un punto y coma: en el primer caso se imprime el resultado de la operación y en el segundo no.

Las variables tienen como limitación que sólo pueden tener valores alfanuméricos, y '_' (comenzando por una letra). Para definir una variable basta con inicializarla de la siguiente manera: *var = <contenido>*. <contenido> puede referir a un número entero (ej: *var = 5*), un decimal (ej: *var = 1.8*), un string (ej: *var = "qwerty"*) o una lista (ej: *var = []*). Una vez inicializada la variable se le pueden empezar a aplicar las operaciones '+', '-', '*' y '/' contra otras variables o nuevos valores. Las variables que se utilizan sin haber sido inicializadas serán consideradas como error sintáctico por el compilador. También se soportan las asignaciones +=, -=, *= y /= siguiendo la siguiente equivalencia: var #= <contenido> es equivalente a var = var # <contenido> (siendo # alguno de los operadores).

Una vez declarada una lista, se le pueden insertar elementos de cualquier tipo al principio de ella con la sintaxis '*<contenido>:var*'. Esta operación es únicamente válida para listas y, como el compilador lleva una tabla con las variables y su tipo en cada contexto, su uso inválido será notificado como error en el compilador.

Existe el operador unario '?' que se aplica a una variable o cualquier valor para conocer su tipo. Por ejemplo se puede aplicar '*50?*', '*12.5*?' o '*var?*', y se imprimirá el nombre del tipo adecuado.

Los bloques if tienen una sintaxis similar a C pero sin llaves y marcando el final del bloque con la palabra reservada 'end:

```
if('expresión booleana')
    código
end
```

Los bloques while también siguen la misma estructura que los if:

```
while('expresión booleana')
    código
end
```

Alternativamente, y en honor al profesor que le da el nombre al lenguaje, la palabras reservadas '*if*' y '*while*' pueden reemplazarse por '*bloque_si*' y '*ciclo_mientras*' respectivamente, siendo así estos bloques equivalentes a los ejemplos de arriba:

```
bloque_si(expresión booleana)
    código
end
ciclo_mientras('expresión booleana')
    código
end
```

En las expresiones booleanas se soportan los comparadores <, <=, >, >=, == y !=, pudiendo comparar entre cualquier tipo de dato. Es importante destacar que muchas de las veces que se comparen dos tipos distintos se retorna false (ej: *1<"2.0"* retorna false pero *1<2.0* retorna true)

5_ Dificultades

Una de las principales dificultades del proyecto fue imprimir el programa compilado en C con orden correcto. Para ello, creamos un struct por cada producción (ver *yaccObjects.h*) creando una estructura de árbol. Cada tipo de struct se sabe imprimir e imprime a sus hijos, por lo que al imprimir el struct 'programa' (*printProg()* en *paco.y*) se imprimen todas las líneas en C recorriendo el árbol generado.

Tuvimos problemas con el scope de las variables, teniendo inicialmente en mente que, como en C, las variables declaradas en un bloque no puedan utilizarse fuera de él, queríamos hacer uso del mismo sistema. A la hora de utilizar o declarar una variable debíamos verificar si se había declarado en un bloque superior o no. De la misma forma, si todavía no se había declarado, debíamos decidir si declararla o no en el código C generado. Debido a que nos trajo muchos problemas decidimos que todas las variables sean globales (todas se definen en el la función *main()*). Sin embargo, el mismo día de la entrega encontramos una solución para resolver este problema sin recurrir a variables globales, pero por la falta de tiempo para su implementación y testeo no la implementamos.

Otra de las dificultades técnicas fue que al querer compilar un programa con el compilador no se detectaba EOF. Finalmente, encontramos que el error estaba relacionado con la función *yywrap()* del parser.

6_ Limitaciones del Proyecto

Una de las principales limitaciones del proyecto es que las listas no son cómodas para acceder a sus valores en una posición específica o para removerlos de la misma, tornándose tedioso realizar programas más complejos.

Además, los cálculos no permiten el uso de los paréntesis, aunque sí se respetan las prioridades operatorias.

Otra limitación a la hora de programar en Paco++ consiste en que las expresiones booleanas dentro de los if y while sólo pueden contener una única condición (no se puede usar el equivalente a && y ||).

Una limitación operativa del lenguaje es que soporta int y float, por lo que no se pueden hacer operaciones con números demasiado grandes porque se llega al overflow fácilmente.

Las listas solo puede agregar objetos, por lo que se dificulta el su uso para ciertas aplicaciones.

7_ Extensiones

Nos hubiese gustado implementar las listas con más funcionalidades, y también las matrices como tipo básicos, para poder realizar operaciones entre matrices y que por ejemplo al multiplicar dos de estos objetos aplique la correspondiente multiplicación de matrices.

El manejo de memoria puede ser mejorado ya que no se liberan todos los recursos usados.

Los tipos int y float podrían ser reemplazados por long y double para obtener mayor rango para las operaciones.

Podrían agregarse otros tipos de errores de compilación además de los presentes '*syntax error*' (mensaje default de yacc) y "*variable not defined*" (mensaje creado por nosotros).