Parseo y Generación de Código – 2do semestre 2019

Licenciatura en Informática con Orientación en Desarrollo de Software

Universidad Nacional de Quilmes

Trabajo práctico 1

Generador de código Avalancha $\longrightarrow$ C++

Fecha de entrega: jueves 14 de noviembre

Índice

1. Introducción

Este TP consiste en implementar un generador de código para el lenguaje Avalancha. La salida será un programa en C++. El compilador de Avalancha implementar las siguientes funcionalidades:

- Construcción de estructuras encabezadas por constructores, p.ej. `Cons(True, Cons(False, Nil))`.
- Invocaciones a funciones, que pueden ser recursivas.
- Resolución del *pattern matching*.
- Verificación de todas las pre y postcondiciones de todas las funciones cada vez que se las invoca.
- Administración automática de memoria, basada en la técnica de *conteo de referencias*.

Para este TP extenderemos la gramática con una palabra clave "`print`":

`print` ⟨PRINT⟩

y con una construcción que permite imprimir el resultado una expresión:

⟨*chequeo*⟩ $\longrightarrow$ ...
| ⟨PRINT⟩ ⟨*expresión*⟩

2. Compilador

El programa generado en C++ se compone de tres partes, que llamamos el *encabezado* del programa, las *funciones definidas por el usuario* y la *función principal* (`main`).

2.1. Encabezado del programa en C++

El compilador debe emitir un *encabezado* del programa que incluye definiciones de estructuras de datos y funciones auxiliares que el compilador provee como parte de la biblioteca necesaria para ejecutar programas en Avalancha. En primer lugar, necesitamos incluir algunos *headers* estándar de C++:

```
#include <vector>
#include <string>
#include <iostream>
using namespace std;
```

Recordemos que el único tipo de datos del lenguaje Avalancha es el tipo de los términos. Por ejemplo, `Cons(True, Cons(False, Nil))` es un término. Para representar un término en C++ utilizaremos la siguiente estructura de datos:

```
typedef int Tag;
struct Term {
  Tag tag;
  vector<Term*> children;
  int refcnt;
};
```

Es decir, un término es un árbol representado utilizando punteros que en cada nodo cuenta con:

- Un `tag` numérico que indica cuál es el constructor que lo encabeza.
- Un vector `children` con punteros a los hijos.
- Un número `refcnt` llamado *contador de referencias* que indica cuántas veces se está utilizando el término en cuestión. Esto servirá para administrar automáticamente la memoria: cuando el contador de referencias desciende a 0, es porque este término ya no se utiliza, y es momento de liberar la memoria.

Como primer paso, el compilador debe hacer un recorrido del código fuente completo del programa para determinar cuáles son **todos** los constructores que se utilizan. Además, se le debe atribuir un tag numérico único a cada constructor. Por ejemplo, para el siguiente programa:

```
fun esNat
  Zero   -> True
  Suc(n) -> esNat(n)
  _      -> False

fun esLista
  Nil         -> True
  Cons(_, xs) -> esLista(xs)
  _           -> False

fun zip : xs, ys -> _
  ? esLista(xs) and esLista(ys)
  Nil         , _           -> Nil
  _           , Nil         -> Nil
  Cons(x, xs) , Cons(y, ys) -> Cons(Par(x, y), zip(xs, ys))
```

los tags podrían ser los siguientes:

Constructor	Tag
`True`	0
`False`	1
`Zero`	2
`Suc`	3
`Nil`	4
`Cons`	5
`Par`	6

Además, como parte del encabezado del programa se deben implementar las tres funciones siguientes:

- `void incref(Term* t);` — Incrementa en 1 el contador de referencias del término `t`. Esta función se puede implementar haciendo sencillamente `t->refcnt++;`.
- `void decref(Term* t);` — Decrementa en 1 el contador de referencias del término `t`. Si el contador de referencias queda en 0, se debe liberar la memoria del término `t`, haciendo `delete t;`. Además, al liberar la memoria se deben decrementar también los contadores de referencias de todos los hijos del término `t`, llamando recursivamente a la función `decref` sobre cada uno de los hijos.
- `void printTerm(Term* t);` — Muestra en pantalla el término `t`, seguido de un fin de línea. Un término se muestra de la siguiente manera:
  - En primer lugar, se muestra el nombre del constructor.
  - Si el término tiene hijos, se muestran recursivamente todos los hijos, encerrados entre paréntesis y separados por comas.
- `bool eqTerms(Term* t1, Term* t2);` — Devuelve verdadero si los términos `t1` y `t2` son iguales (tienen la misma estructura, vistos como árboles).

2.2. Funciones definidas por el usuario

Como siguiente paso, el compilador debe recorrer la lista de declaraciones, es decir, las funciones definidas por el usuario. En primer lugar, el compilador debe asignarle a cada función definida por el usuario un nombre único en C++. El nombre de las funciones en C++ será siempre de la forma `f_<número>`, por ejemplo `f_123`. Además, cada función debe tener asociadas dos funciones `pre_<número>` y `post_<número>` que sirven respectivamente para verificar la pre y la postcondición de la función en cuestión. Por ejemplo, para el programa escrito en Avalancha de la sección anterior, los nombres de las funciones podrían ser:

Nombre (Avalancha)	Función (C++)	Función que verifica la pre	Función que verifica la post
esNat	`f_0`	`pre_0`	`post_0`
esLista	`f_1`	`pre_1`	`post_1`
zip	`f_2`	`pre_2`	`post_2`

A continuación, se debe generar código para las declaraciones (también llamadas "prototipos") de las funciones (respetando sus respectivas aridades). Notar que: los parámetros y el resultado de la función son siempre de tipo `Term*`. Las funciones que verifican la pre y la postcondición no devuelven un resultado (devuelven `void`). La función que verifica la postcondición recibe un parámetro adicional correspondiente al resultado. El nombre del i-ésimo parámetro debe ser `x_i`. Por ejemplo:

```
Term* f_0(Term* x_0);                          // Prototipo de esNat.
void pre_0(Term* x_0);                         // Prototipo de la precondición de esNat.
void post_0(Term* x_0, Term* res);             // Prototipo de la postcondición de esNat.
Term* f_1(Term* x_0);                          // Prototipo de esLista.
void pre_1(Term* x_0);                         // Prototipo de la precondición de esLista.
void post_1(Term* x_0, Term* res);             // Prototipo de la postcondición esLista.
Term* f_2(Term* x_0, Term* x_1);               // Prototipo de zip.
void pre_2(Term* x_0, Term* x_1);              // Prototipo de la precondición de zip.
void post_2(Term* x_0, Term* x_1, Term* res);  // Prototipo de la postcondición de zip.
```

Después de generar los prototipos para todas las funciones, se debe generar la implementación de todas ellas en C++. La implementación de una función `f_<número>` debe:

- Verificar que se cumpla la precondición sobre los parámetros.
- Hacer *pattern matching* para determinar qué rama ejecutar. Si los argumentos no coinciden con ninguno de los patrones, la función devuelve `False`.
- Ejecutar la rama elegida, para calcular un resultado.
- Verificar que se cumpla la postcondición sobre el resultado.

- Devolver el resultado.

Por ejemplo, la implementación de la función `zip` tendrá la siguiente estructura:

```
// Implementacion de zip.
Term* f_2(Term* x_0, Term* x_1) {
  pre_2(x_0, x_1); // Verificar que se cumple la precondición.

  if (/* [ coincide con el patrón (Nil, _) ] */) {
    // [ Guardar Nil en una variable res. ]
    post_2(x_0, x_1, res); // Verificar que se cumple la postcondición.
    return res;
  }

  if (/* [ coincide con el patrón (_, Nil) ] */) {
    // [ Guardar Nil en una variable res. ]
    post_2(x_0, x_1, res); // Verificar que se cumple la postcondición.
    return res;
  }

  if (/* [ coincide con patrón (Cons(x, xs), Cons(y, ys)) ] */) {
    // [ Calcular Cons(Par(x, y), zip(xs, ys)) en una variable res. ]
    post_2(x_0, x_1, res); // Verificar que se cumple la postcondición.
    return res;
  }

  // Si no coincide con ningún patrón:
  // [ Guardar False en una variable res. ]
  post_2(x_0, x_1, res); // Verificar que se cumple la postcondición.
  return res;
}
```

2.3. Función principal en C++

La función principal o `main` de C++, será de la siguiente forma:

```
int main() {
  //
  // [ Código para ejecutar todos los comandos print y check. ]
  //
  return 0;
}
```

2.4. Compilación del comando `print`

Para compilar el comando `print E`, suponiendo que `E` es una expresión, se debe:

- Compilar la expresión `E`, guardando el resultado en una variable temporal `t`.
- Incrementar el contador de referencias de `t`.
- Mostrar el término `t` invocando a la función `printTerm`.
- Decrementar el contador de referencias de `t`.

2.5. Compilación del comando `check`

Para compilar el comando `check F`, suponiendo que `F` es una fórmula, se debe:

- Calcular el valor booleano de `F`.
- Si `F` no es `true`, se debe emitir el mensaje de error `"check failed"` y finalizar inmediatamente la ejecución del programa, llamando a la función `exit(1);`.

2.6. Compilación del mecanismo de pattern matching

Para compilar una función `foo` de aridad n, que consta de m reglas:

```
fun foo
  regla_1
  ...
  regla_m
```

se debe generar código como el siguiente. Supongamos por ejemplo que el número asociado a la función `foo` es el número 42:

```
Term* f_42(Term* x_0, Term* x_1, ..., Term* x_<n-1>) {
  pre_42(x_0, x_1, ..., x_<n-1>);

  if (/* [ (x_0, ..., x_<n-1>) coincide con el patrón de regla_1 ] */) {
    // [ Evaluar el cuerpo de regla_1 y guardarlo en una variable res. ]
    post_42(x_0, x_1, ..., x_<n-1>, res);
    return res;
  }

  ...

  if (/* [ (x_0, ..., x_<n-1>) coincide con el patrón de regla_m ] */) {
    // [ Evaluar el cuerpo de regla_m y guardarlo en una variable res. ]
    post_42(x_0, x_1, ..., x_<n-1>, res);
    return res;
  }

  // Si los parámetros no coinciden con ningún patrón:
  // [ Guardar False en una variable res. ]
  post_42(x_0, x_1, ..., x_<n-1>, res);
  return res;
}
```

Para determinar si la lista de parámetros `(x_0, ..., x_<n-1>)` *coinciden* (es decir, "matchean") contra los patrones `(p_0, ..., p_<n-1>)`, se debe verificar que para todo $i = 0..n-1$ el término `x_i` coincide con el patrón `p_i`.

Para determinar si un término `t` coincide con un patrón `p`, se analizan recursivamente el patrón `p` y el término `t`:

- Si el patrón es una variable, el término `t` siempre coincide con el patrón.
- Si el patrón es un "comodín" representado con un guión bajo (`_`), el término `t` siempre coincide con el patrón.
- Si el patrón es de la forma `C(p_1, ..., p_n)` donde `C` es un constructor y `p_1`, ..., `p_n` son patrones, el término `t` coincidirá con el patrón siempre y cuando `t` sea de la forma `C(t_1, ..., t_n)` y además para todo $i = 1..n$ el término `t_i` coincida con el patrón `p_i`.

Al compilar el cuerpo de una rama, las variables definidas por el patrón deben quedar ligadas al subtérmino correspondiente. (Mirar el ejemplo al final de esta sección).

2.7. Compilación de expresiones

Una expresión `E` se compila recursivamente de la siguiente manera a código en C++ que sirve par generar una variable `t` de tipo `Term*` que contiene el término que resulta de evaluar dicha expresión:

- **Constructores.** Una aplicación de un constructor `C` a n parámetros `C(E_1, ..., E_n)` se compila de la siguiente manera:
  - Compilar recursivamente la expresión `E_1` en una variable `t_1`, e incrementar su contador de referencias.
  - ...
  - Compilar recursivamente la expresión `E_n` en una variable `t_n`, e incrementar su contador de referencias.

- Guardar en una nueva variable `Term* t` un término nuevo, haciendo `new Term`. Su tag se inicializa con el tag correspondiente al constructor `C`. Su vector de hijos se inicializa con los términos `t_1`, ..., `t_n`. Su contador de referencias se inicializa en 0.

- **Variables.** Una variable de Avalancha debe tener asociada una expresión de C++ en el entorno. La variable se compila a dicha expresión.

- **Aplicación de funciones.** Una aplicación de una función `f` a n parámetros `f(E_1, ..., E_n)` se compila de la siguiente manera:

  - Compilar recursivamente la expresión `E_1` en una variable `t_1`, e incrementar su contador de referencias.
  - ...
  - Compilar recursivamente la expresión `E_n` en una variable `t_n`, e incrementar su contador de referencias.
  - Guardar en una nueva variable `Term* t` el resultado de llamar a la función correspondiente a `f` con los n parámetros `t_1`, ..., `t_n`.
  - Decrementar los contadores de referencias de `t_1`, ..., `t_n`.

2.8. Compilación de fórmulas

Una fórmula `F` se compila recursivamente de la siguiente manera a código en C++ que sirve para generar una variable `b` de tipo `bool` que contiene el valor de verdad que resulta de evaluar dicha fórmula:

- **Constantes.** Para compilar la fórmula `true`, guardar en una nueva variable `bool b` la constante `true` de C++. Ídem para `false`.

- **Igualdad.** Si `e_1` y `e_2` son expresiones, la fórmula `e_1 == e_2` se compila de la siguiente manera:

  - Compilar la expresión `e_1` en una variable `t_1`, e incrementar su contador de referencias.
  - Compilar la expresión `e_2` en una variable `t_2`, e incrementar su contador de referencias.
  - Guardar en una nueva variable `bool b` el valor de `eqTerms(t_1, t_2);`.
  - Decrementar los contadores de referencias de `t_1` y `t_2`.

- **Operadores lógicos.** Si `F1` y `F2` son fórmulas, la fórmula `F1 and F2` se compila de la siguiente manera:

```
[guardar el resultado de calcular F1 en b]
if (b) {
  [guardar el resultado de calcular F2 en b]
}
```

  Algo similar para los conectivos `or`, `imp` y `not`. Tener en cuenta que los operadores deben ser de cortocircuito—por ejemplo, en el caso de `(F1 or F2)`, si la fórmula `F1` es verdadera no se debe evaluar la fórmula `F2`, porque el resultado de la disyunción es verdadero.

2.9. Compilación de pre y postcondiciones

Para chequear que se cumplan las pre y postcondiciones, se debe verificar que la correspondiente fórmula sea verdadera. En caso de que la precondición de una función `foo` sea falsa, el programa debe emitir el mensaje de error `pre(foo) failed` y finalizar la ejecución (llamando a `exit(1);`). De igual modo, en caso de que la postcondición sea falsa, el programa debe emitir el mensaje de error `post(foo) failed` y finalizar la ejecución.

2.10. Ejemplo de compilación

La siguiente función en Avalancha:

```
fun ultimo
  Cons(x, Nil) -> x
  Cons(x, xs)  -> ultimo(xs)
```

podría compilar al siguiente código en C++ (o alguno equivalente), teniendo en cuenta que los tags para los constructores son `False` $\mapsto$ `0`, `True` $\mapsto$ `1`, `Cons` $\mapsto$ `2`, `Nil` $\mapsto$ `3`:

```
    Term* f_0(Term* x_0) {
      check_pre_0(x_0);

      if (x_0->tag == 2 && x_0->children[1]->tag == 3) {
        Term* res = x_0->children[0];
        check_post_0(x_0, res);
        return res;
      }

      if (x_0->tag == 2) {
        incref(x_0->children[1]);
        Term* t_2 = f_0(x_0->children[1]);
        decref(x_0->children[1]);
        Term* res = t_2;
        check_post_0(x_0, res);
        return res;
      }

      Term* t_3 = new Term;
      t_3->tag = 0;
      t_3->refcnt = 0;
      Term* res = t_3;
      check_post_0(x_0, res);

      return res;
}
```

3. Pautas de entrega

Para entregar el TP se debe enviar el código fuente por e-mail a la casilla `foones@gmail.com` hasta las 23:59:59 del día estipulado para la entrega, incluyendo `[TP lds-est-parse]` en el asunto y el nombre de los integrantes del grupo en el cuerpo del e-mail. No es necesario hacer un informe sobre el TP, pero se espera que el código sea razonablemente legible. Se debe incluir un README indicando las dependencias y el mecanismo de ejecución recomendado para que el programa provea la funcionalidad pedida. Se recomienda probar el programa con el conjunto de tests provistos.