Instituto Tecnológico de Buenos Aires

Autómatas, Teoría de Lenguajes y Compiladores

Trabajo Práctico Especial

1º cuatrimestre 2021

AUTOR	LEGAJO
Rodríguez, Manuel Joaquín	60258
Arca, Gonzalo	60303
Parma, Manuel Félix	60425

Fecha de entrega: 27/06/2021

Índice

Índice **2**

Idea subyacente y objetivo del lenguaje **4**

Consideraciones realizadas (no previstas en el enunciado) **4**

Tipos de datos 4

Operaciones con datos 5

No se puede elegir un punto de ejecución 5

Funciones personalizadas 6

Impresiones a pantalla por STDOUT 6

Lecturas por STDIN 6

Aclaración sobre read y print 6

Condicionales 6

Ciclos 6

Caracteres especiales 6

Palabras reservadas 7

Descripción del desarrollo del TP **7**

Descripción de la gramática **7**

Declaración y asignación de variables 7

Bloque condicional 8

Declaración de predicados 8

Bloques de repetición hasta incumplimiento de condición 8

Declaración y asignación de máquina 8

Transición 9

Ejecución de maquina 9

Dificultades encontradas **9**

Todos los datos de tipo String se alocan en el heap 9

Lectura de STDIN 10

Scope de variables (machine y predicate son globales y solo pueden llamar variables globales) 10

Palabras, funciones, variables y macros reservadas de C 10

Futuras extensiones, con una breve descripción de la complejidad de cada una **10**

Lectura de STDIN para enteros, booleanos, y char 10

Implementación de arrays para chars, strings e integers. 11

Función que calcula longitud de string 11

Gráficos de autómatas 11

Máquinas y predicados locales 11
Predicates más complejos 11
Mayor soporte para ciclos 12
Return determinístico en main de C 12
Condicionales y ciclos sin bloque 12

Referencias **12**

Idea subyacente y objetivo del lenguaje

Parsium es un lenguaje de programación diseñado para declarar y ejecutar máquinas de estados finitos bajo un paradigma declarativo. Permitiendo al usuario parsear cadenas con el formato que él decida, y tomar decisiones a partir de si son o no aceptadas. El formato preferido para los archivos del lenguaje es `.pars`.

Para esto, el usuario puede definir una máquina que contiene un estado inicial, un conjunto de estados finales y un conjunto de transiciones entre los nodos que componen a dicha máquina. Cada transición tiene asignado un atributo `when` que, en caso de cumplirse, avanzará al estado indicado. `when` puede ser un carácter específico o el nombre de un *predicado*.

Un *predicado* (`predicate` dentro del lenguaje) es una función programada por el usuario que, dado un carácter que recibe por parámetro, retorna un valor booleano.

La máquina que se genera simula un *autómata finito determinístico*, pues a pesar de que un usuario puede asignar a un estado varias transiciones con condiciones equivalentes, se tomará solo la primera de éstas dentro del lenguaje (por ejemplo, se podría asignar a un mismo estado 2 transiciones distintas con la condición de que el carácter actual sea una 'a', y el lenguaje no lo valida y simplemente toma la primera declarada).

Si la máquina se encuentra en un estado y con el carácter actual no puede tomar ninguna de las transiciones definidas por el usuario, se lo llevará al estado ERROR.

Los distintos *predicates* tendrán acceso a las variables globales pero no tendrán acceso a las variables definidas en otros *predicates* o a aquellas que fueron definidas en un bloque más profundo que el global.

Consideraciones realizadas (no previstas en el enunciado)

Tipos de datos

Actualmente el lenguaje soporta los siguientes tipos de datos de forma explícita (es decir, datos que se pueden asignar a variables y declararse mediante la sentencia `def`):

Tipo	Abstracción
`char`	Carácter, idéntico a `char` en C
`int`	Entero, mapeado a un `long` en C

`bool`	Booleano, mapeado a un `bool` en C.
`string`	Cadena de caracteres, mapeado a un `char*` en C. Son null-terminated.
`machine`	Máquina de estados finitos. Contiene las transiciones y estados finales e inicial de la máquina.
`predicate`	Predicado. Simula ser un puntero a una función anónima, que recibe un char y devuelve booleano. Similar a `function_handle` en MATLAB/Octave

Luego, existen también tipos implícitos, utilizados en el "constructor" de `machine`. Éstos serían los tipos de "transición", el de "estado", y los arreglos de estos tipos. Se denominan implícitos ya que no pueden asignarse a variables o declararse formalmente con la sentencia `def`, sino que se utilizan de forma anónima en el constructor de `machine`.

Cabe mencionar que no se pueden declarar datos de tipo `machine` y `predicate` dentro de bloques.

Operaciones con datos

Todos los tipos explícitos de datos soportan al operador `==` y `!=` salvo `machine`. Los datos deben ser del mismo tipo a izquierda y a derecha del operador para que sea aceptado por el lenguaje. La igualdad de strings es igual que la de C, es decir, compara los punteros al comienzo de las cadenas.

Sólo los caracteres y los enteros soportan los operadores `< <= > >=` para comparar valores. En caracteres se compara por valor en la tabla ASCII, y en enteros por valor del entero.

A diferencia de C, el único tipo dato que cobra sentido como booleano es el de tipo `bool`. Es decir que intentar realizar operaciones lógicas con tipos de datos distintos de `bool` no es aceptado en el lenguaje.

La sentencia `return` sólo soporta valores de tipo `bool` ya que sólo utilizamos `return` dentro de los predicados y en el hilo principal de ejecución (corta el proceso al igual que `return` en el `main` de C).

No se puede elegir un punto de ejecución

El punto de ejecución en el lenguaje es siempre la primera línea en el código, no hace falta declararlo de ninguna manera.

Funciones personalizadas

La única forma de utilizar funciones creadas por el usuario dentro del lenguaje es a través de la creación de predicados, que pueden ser creados por el usuario y luego llamados en cualquier parte del código (devuelve `bool`) o asignados a las transiciones de las máquinas.

Impresiones a pantalla por STDOUT

Dentro del lenguaje está disponible la función `print()`, que permite imprimir una constante o una variable de tipo `int`, `char`, `string`, y `bool` a STDOUT. Devuelve la cantidad de caracteres impresos.

Lecturas por STDIN

Existe la función `read()` que permite leer de entrada estándar y asignar el valor leído a un string. No permite conversiones a datos que no sean strings (es decir, no existe una función que, por ejemplo, lea de STDIN y convierta lo leído a un entero). Adicionalmente, soporta lecturas de hasta 8095 bytes.

Aclaración sobre read y print

En cada llamado a las funciones `read()` y `print()` se vacía el buffer de STDIN y de STDOUT, respectivamente.

Condicionales

El lenguaje soporta la sentencia "`if (condition) {} else {}`", sin embargo no soporta `if` ni `else` en una sola línea (es decir, sin llaves que abran), `else if () {}`, ni el operador ternario del estilo "`condition ? a : b`".

Ciclos

El proyecto soporta la sentencia `while(condition) {}` para ciclos, pero no soporta `do-while`, `for`, `for-each`. o `while` en una sola línea,

Caracteres especiales

Para que el analizador léxico levante caracteres especiales debimos generar reglas específicas pues no validan con la expresión regular de un carácter: '`.`'. Por esto, se decidió soportar los que consideramos de mayor importancia, estos son: '`\n`', '`\r`' y '`\t`'.

Palabras reservadas

_____Además de las palabras utilizadas por la gramática del lenguaje, se asume que no se hará uso de las palabras "`NULL, N, read_tmp_00, read_tmp_01, read_tmp_02`", ya que se utilizan internamente en C. Estas últimas palabras son atajadas por el compilador. Sin embargo, cualquier otra palabra que sea reservada de C tampoco debería ser utilizada por el usuario, y éstas no son validadas, por lo cual pueden haber errores de GCC en momento de compilar código que viole tal regla. Más sobre esto en Dificultades encontradas.

Descripción del desarrollo del TP

El compilador funciona con un analizador léxico que utiliza Lex como scanner y un analizador sintáctico basado en un parser que funciona sobre el lenguaje Yacc.

Durante el desarrollo del analizador léxico se tuvo cuidado de que las reglas sean completamente excluyentes para que todas tengan el comportamiento esperado y no existan reglas inaccesibles.

Y en la parte del analizador sintáctico se tuvo especial cuidado con que no exista ambigüedad en el parser generado, es decir, que no existan conflictos del tipo shift-reduce o reduce-reduce, pues esto conlleva a un comportamiento indeterminado por parte del mismo.

Luego de la parte de análisis, que construye un *Abstract Syntax Tree*, se realiza la traducción del mismo al lenguaje C teniendo los cuidados necesarios para asegurar que se realice una compilación segura. El AST nos permitió mantener un muy buen manejo de toda la transformación.

Junto al AST, se mantiene un array donde se encuentran todas las máquinas definidas y otro array donde se encuentran todos los predicates definidos. Lo que nos permite en la traducción ponerlos antes que cualquier ejecución y así asegurarnos la correcta compilación.

Descripción de la gramática

El lenguaje compilará todas las instrucciones en el orden en que se las escribe. Se utiliza como carácter delimitador al ‘;’.

Declaración y asignación de variables

```
def TYPE varId = expression;
```

Donde TYPE puede ser cualquiera de los siguientes: *char*, *bool*, *machine*, *predicate*, *int* o *string*. Y *expression* debe retornar un valor del mismo tipo que la variable declarada.

El lenguaje también soporta solo declaración y solo asignación(de una variable ya declarada anteriormente).

Bloque condicional

```
if ( condition ) {
      block
} else {
      block
}
```

Recordamos que el `else` y su respectivo bloque son opcionales.

Declaración de predicados

```
def predicate predicateId = ( param_name ) {
      block
};
```

Existen tres predicates ya definidos para que el usuario pueda utilizar, estos son:

`isNumber` : devuelve true si el carácter actual es un número.
`isUpperCase` : devuelve true si el carácter actual es una mayúscula.
`isLowerCase` : devuelve true si el carácter actual es una minúscula.

Bloques de repetición hasta incumplimiento de condición

```
while( condition ) {
      block
}
```

Declaración y asignación de máquina

```
def machine machineId = <<
      transitions = [ transitionsArray ],
      initialState = initialStateId,
```

```
      finalStates = [ finalStatesIdArray ]
>>;
```

Transición

```
originSymbol -> destinationSymbol when condition
```

Donde *condition* puede ser:

`ANY` : sin importar el carácter actual se tomará esta transición.

'`carácter`' : se tomará la transición sólo si el carácter actual es el que se indica en la condición.

`predicateId` : se tomará la transición sólo si la ejecución del predicate devuelve *true*.

Ejecución de maquina

```
parse string with machineId;
```

Esta ejecución retornara un boolean dependiendo de si se aceptó o no la cadena (ya sea explicita o mediante una variable) en la máquina indicada. Este valor de retorno se puede asignar a una variable boolean.

Dificultades encontradas

Todos los datos de tipo String se alocan en el heap

Al implementar la lectura desde STDIN en nuestro lenguaje, nos encontramos con el problema de que la función read() debería admitir toda variable de tipo String. En C, los "strings" son en realidad arrays de caracteres, y cualquier string definido como constante en el código no puede ser modificado. Por lo tanto, se decidió que la declaración de una variable de tipo String en nuestro lenguaje alocara espacio de memoria en el heap, y que la asignación se realice con copia de caracteres. También se decidió esto ya que debería ser posible en nuestro lenguaje la asignación directa de String desde constantes como desde variables. Si no se alocara la memoria para luego copiar los caracteres con funciones de la librería estándar de C, serían imposibles estas asignaciones.

Lectura de STDIN

Implementar `read()` para datos de tipo `int`, `char` y `bool` no fue posible, ya que actualmente la función `read()`, que sirve para leer un string de entrada estándar, devuelve un string, el cual no requiere validación. Sin embargo los otros tipos pueden recibir entrada inválida, y en este caso se precisaría de un valor de retorno que refleje el caso de error, el cual no tenemos ya que no se utilizan punteros para los tipos `int`, `char` y `bool`. Por lo tanto, se decidió utilizar únicamente lectura de tipo String.

Scope de variables (machine y predicate son globales y solo pueden llamar variables globales)

Al no tener un punto de entrada en nuestro lenguaje, nos encontramos con el problema del scope de las variables para C. Con esto nos referimos a cómo elegir dónde se definirán las variables en la compilación de nuestro lenguaje a código C. Se decidió que las variables de tipo `Machine` y las funciones `Predicate` sean globales y no se permitan definir dentro de bloques en nuestro lenguaje. Por otro lado, las variables de los otros tipos que no se definan en un bloque tendrán declaración global en C (para poder utilizarlas dentro de `Predicates`) pero su asignación se realizará dentro de la función `main()` de C, para permitir asignación de resultados de funciones (como `Parse` o `Predicate`).

Palabras, funciones, variables y macros reservadas de C

Debido al código final generado en C, se debió prohibir el uso de ciertas palabras en nuestro lenguaje. Por un lado, algunas macros utilizadas para desarrollar la máquina de estados (como “N” y “NO_CHAR”), así como macros de la librería estándar de C (como “NULL”). Si no estuvieran prohibidas, se podría tener una variable de nombre “NULL” u otros casos similares.

Futuras extensiones, con una breve descripción de la complejidad de cada una

---

Lectura de STDIN para enteros, booleanos, y char

Cómo se indicó en dificultades, sólo se permite la lectura de tipo String desde STDIN, y no de `int`, `char` y `bool`. Una posible solución sería que dichos tipos soporten el valor NULL, pero esto implica refactorizaciones importantes en gran parte del código. Debido a esto, se considera que el problema es de dificultad media/alta.

Implementación de arrays para chars, strings e integers.

Baja dificultad, gramaticalmente es un comportamiento similar al del array de transiciones o de estados finales que si se encuentran implementados.

Función que calcula longitud de string

Le agregaría valor al usuario para que pueda realizar implementaciones de predicados cuyas condiciones involucren la longitud de un string determinado. Es de fácil implementación pues sería realizar una traducción a la función `strlen` ya existente en C, aunque se debería tener cuidado con que no se la invoque con un parámetro de tipo distinto a string.

Gráficos de autómatas

Se realizaría con una traducción al lenguaje GraphViz (https://graphviz.org/), y ejecución del mismo archivo, para obtener una imagen que muestra el grafo representativo de la máquina. Sería de una dificultad media debido a que por más que sea una traducción fácil, habría que resolver cómo diagramar el hecho de que solo se tome la primera transición válida en caso de haber muchas posibles.

Máquinas y predicados locales

Debido a que la traducción a C de una máquina y de un predicado conlleva la construcción de una estructura global y de una función respectivamente, no se pudo implementar la localidad de los mismos. Consideramos que la implementación de esto sería de gran dificultad debido a que se debería hacer un mejor manejo de parámetros, y C no permite la creación de funciones de manera no global.

Predicates más complejos

Predicados que permitan el manejo de múltiples parámetros de tipo distinto a char permiten agregarle gran potencia al lenguaje. Esto sería de gran dificultad, por el manejo de las validaciones de que estos se llaman correctamente y que se realice una traducción correcta.

Mayor soporte para condicionales

El lenguaje no soporta la sentencia `else if` ni la condición ternaria (por ejemplo: `(a)? b : c`) que son de gran valor para el usuario al ser muy populares en los lenguajes actuales. En el caso de la primera sentencia creemos que no sería de fácil implementación pues podría llevar a una concatenación infinita de `else if`, la cual se debería soportar o restringir de alguna forma. En cuanto a la otra expresión condicional

pensamos que sería fácil debido a que sería cuestión de agregar ciertas reglas en la gramática y cuidados en la traducción.

Mayor soporte para ciclos

Actualmente sólo permitimos el uso de la sentencia `while` para manejo de ciclos, por lo que se podrían implementar los ciclos `for` y `do-while`.

Return determinístico en main de C

Nuestro compilador no requiere del uso de un `return` para insertar al final de la función `main()` en C, por lo que la salida del programa dependerá del compilador de C utilizado si el usuario no corre un return al final código.

Condicionales y ciclos sin bloque

Sólo permitimos el uso de `if` y `when` cuando le sigue un bloque abriendo y cerrando con llaves, por lo que se podría implementar el uso de estos para instrucciones de una única línea.

Referencias

---

1. Implementación de scope para variables: https://stackoverflow.com/a/66132901
2. John R. Levine, Tony Mason, Doug Brow - “Lex & Yacc” - Octubre, 1992