DISEÑO DE COMPILADORES

PROYECTO FINAL

Equipo:

[A01128633] Guillermo Beltrán Gómez
[A01128794] Julio Gerardo Collado López
[A01128621] Arturo Moysen Badillo

Profesora: Maricela Quintana López

Fecha: 30 de Noviembre de 2013

Campus: Cuernavaca

Índice

Contenidos

Índice 0
Introducción 2
Herramientas utilizadas 2
Definición del lenguaje 2
Tipos de Datos 2
Identificadores 3
Variables 3
Estructura de Datos Heterogéneos 3
Arreglos 4
Operadores Aritméticos y Expresiones Aritméticas 4
Operadores Lógicos y de Relación 5
Expresiones Lógicas 5
Estructuras de Condición 5
Estructuras Iterativas 6
Estatuto Compuesto 6
Funciones y Firmas 7
Estatutos de entrada y salida 7
Comentarios 7
Estructura de un programa 8
Ejemplo 8
Conclusiones 9
Referencias 9

Introducción

El objetivo de este proyecto final fue desarrollar un compilador para un lenguaje de programación básico llamado “JaguarC” (mismo que fue creado por nosotros), siendo capaz de detectar errores en el código de un programa e indicar el tipo de error y la línea en la que se originó.

El lenguaje tiene un alcance de detección de errores hasta nivel semántico, por lo que no produce código intermedio ni maneja localidades de memoria ni valores de las variables.

Herramientas utilizadas

- **Flex:** Es una herramienta utilizada para generar *scanners* (también conocidos como *tokenizers*), el cuál es un programa que reconoce patrones léxicos en texto, a través de un archivo que contenga la descripción de reglas a través de expresiones regulares y código en C. A partir de esto, Flex genera un código fuente en C con terminación “.yy.c”, el cuál puede ser compilado y utilizado para producir un ejecutable, quien será el encargado de leer una entrada de datos, comparándola con cada regla, y en caso de encontrar alguna, ejecutar el código C correspondiente (The Flex Project, 2008).

- **Bison:** Es una herramienta que convierte una gramática libre de contexto en un *parse*r determinístico LR o generalizado LR en código C, el cual se encarga de leer secuencias de *tokens* y ayuda a decidir si la secuencia cumple con la sintaxis especificada por la gramática (Free Software Foundation, 2008).

Definición del lenguaje

El lenguaje de programación “JaguarC”, cumple con los siguientes aspectos

Tipos de Datos

- **int:** Valores numéricos sin valor decimal
- **float:** Valores numéricos con valor decimal
- **string:** Secuencia de símbolos alfanuméricos
- **bool:** Valor verdadero o falso
- **void:** Valor vacio para funciones.

Identificadores

Un identificador es un nombre único proporcionado por el usuario, y que no se encuentra dentro de las palabras reservadas utilizadas por el lenguaje, mismo que está sujeto a las siguientes condiciones:

- Un identificador sólo puede contener letras, números o el carácter especial "_" (guion bajo)
- El primer carácter en el identificador debe ser una letra o un guion bajo, por ejemplo: *_identificador*, *identificador*
- Un identificador solo puede ser declarado una vez en todo el programa, no importa si se intenta declarar en un ámbito/scope diferente. El nombre del identificador es único en todo el código.

Variables

Se encarga de relacionar un identificador con un valor, cumpliendo con una estructura de inicialización en la forma:

```
tipo identificador;
```

Donde tipo se refiere al tipo de dato que tendrá el valor asociado a la variable. Una variable siempre debe ser definida de esta forma antes de ser utilizada, y su valor debe ser asignado posteriormente de la siguiente forma:

```
identificador = expresión;
```

Donde expresión se refiere a la asignación de un valor, ya sea de forma directa o a través de una expresión aritmética, una función u otra variable.

Estructura de Datos Heterogéneos

Una estructura se identifica por la palabra reservada `struct`, y se define de la siguiente forma

```
struct identificador {
    variables
};
```

Donde variables se refiere a la definición de una o más variables.

Una estructura y sus variables en teoría tienen un ámbito global, ya que pueden ser utilizadas desde cualquier función o del main.

Para utilizar una variable que fue definida dentro de una estructura, se declaran de la siguiente manera:

```
idenfiticador.variable
```

Arreglos

Se permite el uso de arreglos de una dimensión utilizando la siguiente estructura de definición

```
tipo identificador[tamaño];
```

Donde tamaño indica el número de espacios disponibles dentro del arreglo. Siempre debe de realizarse la declaración de un arreglo de esta forma antes de ser utilizado.

Para utilizar un arreglo es necesario pasar dentro de los corchetes un índice, que representa el espacio del arreglo que se desea acceder.

```
identificador[1];
```

En caso de sobrepasar los límites definidos para el arreglo se marcará el error correspondiente.

Operadores Aritméticos y Expresiones Aritméticas

El lenguaje soporta el uso de los operadores aritméticos de suma (+), resta (-), multiplicación (*) y división (/).

Para utilizar una expresión aritmética, se debe cumplir con el siguiente formato:

```
expresión_aritmética + expresión_aritmética
```

Donde una expresión_aritmética puede ser un valor, un identificador de una variable u otro conjunto de expresiones aritméticas, donde el valor resultante será definido por la combinación de valores utilizados en la expresión, en base a las siguientes tablas:

+ (Suma)	String	Int	Float	Bool
String	String	String	String	Error
Int	String	Int	Float	Error
Float	String	Float	Float	Error
Bool	Error	Error	Error	Bool

- (Resta)	String	Int	Float	Bool
String	Error	Error	Error	Error
Int	Error	Int	Real	Error
Float	Error	Real	Real	Error
Bool	Error	Error	Error	Bool

* (Multiplicación)	String	Int	Float	Bool
String	Error	String	Error	Error
Int	String	Int	Float	Error
Float	Error	Float	Float	Error
Bool	Error	Error	Error	Error

/ (División)	String	Int	Float	Bool
String	Error	Error	Error	Error
Int	Error	Real	Real	Error
Float	Error	Real	Real	Error
Bool	Error	Error	Error	Error

Operadores Lógicos y de Relación

- **&&** (y): Relaciona dos valores de tipo bool
- **==** (igual): Compara dos valores, buscando igualdad
- **|** (o): Compara el resultado entre dos valores de tipo bool, buscando que alguno sea verdadero
- **>** (Mayor): Verifica que un valor sea mayor a otro
- **<** (Menor): Verifica que un valor sea menor a otro
- **<=** (menor o igual): Verifica que un valor sea menor o igual a otro
- **>=** (mayor o igual): Verifica que un valor sea mayor o igual a otro

Expresiones Lógicas

Se pueden realizar expresiones lógicas, de la forma:

```
expresión operador_lógico expresión
```

Donde expresión se refiere a una variable o valor de tipo booleano.

Estructuras de Condición

Se utilizan las palabras reservadas IF y ELSE para establecer condiciones de tipo si-entonces, utilizadas de la forma:

```
IF (lista_condiciones) {
```

```
}
ELSE {
}
```

Donde el else es opcional y la lista_condiciones se refiere a una serie de expresiones de tipo bool que deben cumplirse y que deben ser separadas por un operador lógico. Las expresiones son evaluaciones realizadas para determinar un resultado que puede ser verdadero o falso.

Toda condicion dentro de la lista de condiciones debe ser separada entre parentesis; por ejemplo "(condicion)", sin importar si hay una o más condiciones dentro de la lista.

Estructuras Iterativas

Por medio de la palabra reservada for se pueden realizar iteraciones de comandos hasta que una condición de paro se cumpla. Se define de la forma:

```
for(asignación ; condiciones; asignación){
        }
```

Donde asignación se refiere a la asignación de un valor a la variable que se utiliza para evaluar la condición de paro en la parte de condiciones, a través de una expresión lógica. Por ejemplo:

```
int i;
for(i = 0; i < 10; i=i+1){  }
```

En nuestro lenguaje, la primera asignación dentro del for es opcional.

Estatuto Compuesto

Definido mediante los símbolos de { y }, siempre se debe utilizar en pares, es decir, siempre que se declara un { para abrir un estatuto compuesto, éste debe indicar su terminación utilizando }.

Se refiere a una colección de estatutos que deben realizarse tras cumplirse el acceso al estatuto compuesto, esto puede ser mediante una estructura de condición, de iteración o una función.

Funciones y Firmas

Se permite el uso de funciones para realizar un estatuto compuesto. Se definen de la siguiente forma:

```
tipo identificador(parámetros);
```

Donde parámetros se refiere a una o más declaraciones de variables, las cuales pueden ser utilizadas dentro de dicha función. Esta declaración de una función es conocida como una **firma**, la cual es requerida definir antes de utilizar o componer la función.

Por componer la función nos referimos a asignar una lista de sentencias dentro del cuerpo de la función.

Sin embargo si una función es firmada pero no se define su cuerpo o su lista de sentencias no deberá haber error alguno.

Una vez declarada, una función puede ser utilizada de la siguiente forma:

```
identificador(parámetros);
```

Al llamar una función siempre es indispensable pasar como parámetros a la funcion el mismo número de parámetos y que estos sean del mismo tipo que la función está esperando.

De igual manera, al llamar una función se limita a que los parámetros sean necesariamente ID's por lo que no se pueden pasar cadenas, números o booleanos directamente a la función.

Estatutos de entrada y salida

Se pueden utilizar el estatuto de write para mostrar algún valor, y read para obtener input del usuario a través del teclado. Se utilizan de la forma:

```
write(valor);

read();
```

Estos son algunos métodos ya predefinidos en el lenguaje. Write teoricamente imprimiría en pantalla el valor pasado como parámetro. Read regresa un buffer que recibe del teclado del usuario por lo que es necesario recibirlo en una variable tipo string.

Comentarios

Se pueden utilizar comentarios de una sola línea, utilizando la estructura de:

```
//esto es un comentario
```

Todas las lineas que sean comentadas serán ignoradas al momento de tokenizar el código fuente.

Estructura de un programa

Un programa escrito en el lenguaje de JaguarC, considera la siguiente estructura:

Sección	Descripción
program nombre	Indica el inicio y nombre del programa
Estructuras	Sección donde se declaran todas las estructuras ocupadas dentro del programa
Firmas	Las firmas de las funciones del programa; opcionales, ya que podría no haber firmas y/o funciones.
Main	Función principal del programa, cuyo nombre es "main"
Funciones	Todas las funciones que sean firmadas.
programEnd	Indica el fin de un programa

Ejemplo

```
program miprograma:

struct a{
	int y;
};

struct vaso{
	int m[10];
};

//single line comment
int fcn1(string g, int qwe, int wer, string bnm);

main(){
	int ___Jk1;
	int iop[5];
	___Jk1 = iop[4];
	vaso.m[0] = ___Jk1;
	bool malz;
	malz = false;
	if((malz)){
		if((malz!=false)&&(true)||(1<=100)){
			int i;
			i=0;
			for(;(i<10);i=1+1){
				write("Hola mundo");
```

```
                }
            }else{
                if(("hola"<=(1+"mundo"))){
                    string baca;
                }else{
                }
            }
            ___Jk1 = fcn1(___Jk1,ma1z);
        }
}

int fcn1(int qwe, bool wer){
        int mouse;
        return mouse;
}

programEnd
```

Conclusiones

Mediante la elaboración de este proyecto, logramos entender la estructura, proceso de elaboración y herramientas para necesarias para crear un compilador, así como un lenguaje de programación creado por nosotros para probar dicho compilador, obteniendo como resultado un compilador capaz de detectar si un programa se encuentra escrito correctamente o no en “JaguarC” e indicar los errores que se obtuvieron en caso de que existan.

Referencias

The Flex Project (2008). *Overview of Flex*. Recuperado de http://flex.sourceforge.net/

Free Software Foundation (2008). *Introduction to Bison*. Recuperado de http://www.gnu.org/software/bison/