Universidad Nacional Autónoma de México
Facultad de Ingeniería
Compiladores
Semetre 2015-1
Proyecto Final

1. Objetivo

Elaborar un compilador para el lenguaje **C −−** (descrito en la sección de gramática), que realice la traducción a código objeto para una máquina hipotética descrita en la sección de arquitectura de este documento. El compilador será programado usando JFlex(flex) y BYACC/J(yacc).

2. Características del Lenguaje.

Esta sección se describen los tokens válidos para el lenguaje C−− y que serán retornados por el *Analizador Léxico(scanner)*.

2.1. Constantes de Literales

Las constantes de literales para el lenguaje **C−−**, son constantes que incluyen a las cadenas y los caracteres. Una constante de cadena es cualquier concatenación de uno o más caracteres ASCII, encerrados entre comillas dobles. Un caracter es un símbolo ASCII encerrado entre comillas simples.

Ejemplos
Cadenas: "*esto es una cadena*", "*cadena*", "*a*"
Caracteres: '*a*', '*c*'

2.2. Operadores y Delimitadores

2.2.1. Delimitadores

Los delimitadores son símbolos que nos sirven para tener control sobre la estructura del programa y tambíen para agrupar ciertos tipos de tokens. Los delimitadores usados por C−− son los siguientes:

- : " ( ) ; , { } : ' [ ]

2.2.2. Operadores

Los operadores son los siguientes:

- *Operadores aritméticos*: +(suma), -(resta), *(multiplicación), /(división), %(módulo) y -(menos unario).
- *Operadores relacionales*: >(mayor), <(menor), <=(menorigual), >=(mayorigual), ==(*igual*) y ! =(distinto).
- *Operadores lógicos*: ||(or), &&(and), y !(not).

- *Operadores de asignación*: =, +=, -=, *=, /=, %=.

En la tabla siguiente se presenta la precedencia de operadores de mayor a menor, los operadores de la misma línea tienen la misma precedencia

Precedencia	**Asociatividad**
() []	izquierda
++ −− !	derecha
* / %	izquierda
+ -	izquierda
< <= > >=	izquierda
== !=	izquierda
&&	izquierda
||	izquierda
= += -= /= *= %=	derecha

2.3. Identificadores y palabras reservadas.

2.3.1. Identificadores

Un identificador consiste en una secuencia de caracteres, dígitos o guiones bajos, que comienzan por una letra o un guión bajo. Los identificadores sirven para dar nombre a variables o subprogramas, el lenguaje es sensible a las mayúsculas y minúsculas.

2.3.2. Parlabras reservadas

Las palabras reservadas que no pueden asignarse a un identificador son: **int, float, char, double, void, for, while, if, else, case, switch, break, do, default, scan, print, return y struct**.

2.4. Tipos de Datos

Los datos pueden ser del tipo caracter, entero, flotante y doble precisión. En la siguiente tabla se muestra el tamaño de los datos.

Tipo	**Longitud**
char	1 byte
int	2 bytes
float	4 bytes
double	8 bytes

Rango de los tipos de datos

Tipo	**Rango**
char	-128 a 127
int	-32768 a 32767
float	3.4E-38 a 3.4e+38
double	1.7E-308 a 1.7E+308

Las constantes numéricas tienen la misma forma que en el lenguaje ANSI C.

2.4.1. Datos Compuestos

También hay tipos de datos compuestos mediante el uso de strcut, estos tipos de datos pueden estar compuestos por variables de los distintos tipos básicos, así como de arreglos, y otros tipos compuestos. Su longitud se determina por el tamaño de cada uno de los miembros que los componen.

2.5. Sentencias de control de flujo

Se consideran cinco tipos de sentencias.

- **if**(condicion) sentencias1 **else** sentencias2
- **while**(condicion) setencias
- **for**(asignacion; condicion; incremento) sentencias
- **do** sentencias **while**(condicion);
- **switch**(id) casos

Si hay un break dentro de un bucle se debe interrumpir la ejecución del bloque.

2.6. Instrucciones de entrada y salida.

Para la impresión de la salida se cuenta con la instrucción **print** que se encargara de mandar a pantalla tanto los enteros, flotantes, doble precisión, caracteres y cadenas de texto.

Para la lectura desde teclado se cuenta con la instrucción **scan** que se encargara de leer del teclado todos los datos.

2.7. Declaración de Variables.

Se permite la declaración conjunta de variables. Ejemplo: **int x, y, z;**

Además se permite declarar variables dentro de los subprogramas. Estas variables se conocen como variables locales y estas variables deben declararse al principio del subprograma.

2.8. Asignación

La asignación permite del lado derecho también llamadas a funciones.

2.9. Comentarios

Se permite la inclusión de dos tipos de comentarios:
Comentarios de una sola línea //Esto es un comentario de una línea
Comentarios de varias lineas /* Esto es un comentario */

Gramática del Lenguaje C−−

1. programa → lista_declaraciones
2. lista_declaraciones → lista_declaraciones declaracion | declaracion
3. declaracion → declaracion_variables | declaracion_funcion
4. declaracion_variables → tipo lista_variables**;**
5. lista_variables → lista_variables **,** lista | lista
6. lista → **id** | **id**[**entero**]
7. tipo → **int** | **float** | **double** | **char** | **void** | tipo_struct
8. tipo_struct → **struct id {** cuerpo_struct **}** | **struct {** cuerpo_struct **}** | **struct id**
9. cuerpo_struct → cuerpo_struct declaracion_variables | declaracion_variables
10. declaracion_funcion → tipo **id(** parametros **)** bloque
11. parametros → lista_parametros | **void**
12. lista_parametros → lista_parametros **,** parametro | parametro
13. parametro → tipo **id**[ ] | tipo **id**
14. bloque → **{** delcaraciones_locales lista_sentencias **}**
15. declaraciones_locales → declaraciones_locales declaracion_variables | ε
16. lista_sentencias → lista_sentencias sentencia | ε
17. sentencia → sentencia_exp | sentencia_if | sentencia_while | sentencia_do | sentencia_switch | sentencia_for | sentencia_break | sentencia_return | bloque | sentencia_imprime | sentencia_lee
18. sentencia_exp → expresion**;** | **;**
19. sentencia_if → **if(** expresion **)** sentencia sentencia_else
20. sentencia_else → **else** sentencia | ε
21. sentencia_while→ **while(** expresion **)** sentencia
22. sentencia_do → **do** sentencia **while(**expresion **);**
23. sentencia_switch → **switch(id){** lista_casos case_default **}**
24. lista_casos → lista_casos sentencia_case | sentencia_case
25. sentecia_case → **case entero:** sentencia sentencia_break
26. case_defualt → **default:** sentencia sentencia_break | ε
27. sentencia_for → **for(** expresion **;** expresion **;** sentencia_incremento **)** sentencia

28. sentencia_break → **break;**
29. sentencia_incremento → **id++** | **id- -**
30. sentencia_imprime → **print(** expresion **);**
31. sentencia_lee → **scan(id);**
32. sentencia_return → **return;** | **return** expresion **;**
33. expresion → variable operador_asignacion expresion | expresion_simple
34. operador_asignacion → = | += | -= | *= | /= | %=
35. variable → **id** | **id**[expresion]
36. expresion_simple → expresion_simple operador_relacional operando | expresion_simple operador_logico operando | operando
37. operado_logico → && | ||
38. operador_relacional → <= | >= | < | > | == | ! =
39. operando → operando operador_adicion termino | termino
40. oprador_adicion → + | -
41. termino → termino operador_mul factor | factor
42. operador_mul → * | / | %
43. factor → variable | llamada | (expresion) | **entero** | **flotante** | **caracter** | **cadena** | **doubleprecision** | **!**expresion | **-**expresion
44. llamada → **id(** argumentos)
45. argumentos → lista_argumentos | ε
46. lista_argumentos → lista_argumentos **,** expresion | expresion

Consideraciones Semánticas

1. Un programa se compone de una lista de declaraciones, las cuales pueden ser de dos tipos: de variables o de funciones. Debe al menos existir una declaración. Las restricciones semánticas son como sigue, todas las variables y funciones deben ser declaradas antes de utilizarlas. La última declaración de un programa debe ser una función main. Este lenguaje carece de prototipos de funciones, de manera que se hace distinción entre declaraciones y definiciones.

2. La declaración de variables puede ser una variable simple, una variable de arreglo, cuya dirección base es 0 y el último elemento es NUM-1, o bien un conjunto de datos agrupados en un elemento de programación llamado estructura. Las variables pueden ser del tipo int, float, double, char, el tipo void solo se usa para funciones.

3. Las variables presentan una restricción adicional, queda prohibido el uso de apuntadores en C− -.

4. La *Tabla de Símbolos* debe guardar la siguiente información: {Identificador, Tipo de Símbolo, Apuntador a la *Tabla de Tipos*, No. de parámetros, Lista Paramétros, Dirección}. Además en cada ámbito se crea una nueva *Tabla de Símbolos* y se destruye al salir, se puede usar una pila de *Tablas de Símbolos*.

5. Una cadena también puede ser asignada a un arreglo del tipo char que contenga las localidades para almacenar la cadena.

6. Se debe construir una tabla de cadenas que funciona de la misma manera que la tabla de símbolos, excepto que ésta guarda caracteres y cadenas.

7. La declaración de funciones consta del tipo de retorno, un identificador y una lista de parámetros separados por comas dentro de paréntesis, seguidos de una secuencia de código de la función. Si el tipo de retorno es void, la función no devuelve valor alguno. Los parámetros también pueden ser void; es decir no recibe parámetros. Los parámetros seguidos de corchetes, representan arreglos cuyo tamaño puede variar. No hay parámetros del tipo función. Las funciones pueden ser recursivas. Las declaraciones locales tienen un ámbito de existencia igual al de la lista de sentencias del bloque y reemplazan cualquier declaración global.

8. Una *llamada de función* consta de un identificador, seguido por sus argumentos encerrados entre paréntesis. Los argumentos pueden estar vacíos o estar compuestos por una lista de expresiones separadas mediante comas, que representan los valores que se asignarán a los parámetros durante una llamada. Las funciones deben ser declaradas antes de llamarlas, y el número de parámetros en una delcaración debe ser igual al número de argumentos en una llamada. Un parámetro de arreglos en una declaración de función debe coincidir con una expresión compuesta de un identificador simple que representa una variable de arreglo.

9. Un bloque de sentencias se compone de llaves que abren y cierran y de un conjunto de declaraciones y sentencias. Un bloque de sentencia se realiza al ejecutar la secuencia de sentencias en el orden indicado por el programa fuente. Las declaraciones locales tienen un ámbito igual al de la lista de sentencias del bloque de sentencias y reemplaza cualquier declaración global.

10. Se puede dar el caso en que la lista de declaraciones y la lista de sentencias estén vacías.

11. Una sentencia de expresión tiene una expresión opcional seguida por un signo de punto y coma. Tales expresiones por lo regular son evaluadas por sus efectos colaterales. Por consiguiente, esta sentencia se utiliza para asignaciones y llamadas a funciones.

12. La sentencia if tiene la semántica habitual; la expresión evaluada; un valor distinto de cero provoca la ejecución de la primera sentencia; un valor de cero ocasiona la ejecución de la segunda sentencia, si es que existe.

13. La sentencia switch evalua cada uno de los casos para saber cual es el código que se tiene que ejecutar, si se incluye el caso default, éste se debe ejecutar, sino hubo coincidencia con los anteriores casos. switch solo acepta identificadores del tipo int.

14. La sentencia while al evaluar de manera repetida la expresión y al ejecutar entonces la sentencia si la expresión evalúa un valor distinto de cero, finalizando cuando la expresión se evalúa a cero. De la misma manera para el do-while, solo que en este caso se ejecuta la sentencia al menos una vez.

15. Una sentencia de retorno puede o no devolver un valor. Las funciones no declaradas como void deben devolver valores. Las funciones declaradas como void no deben devolver valores. Un retorno provoca la transferencia del control de regreso al elemento que llama.

16. Una sentencia print mandará a imprimir en pantalla ya sea una cadena de texto, o en su caso el valor del resultado al evaluar una expresión.

17. La sentencia break, cuando esta relacionada con un ciclo while, deberá de terminar la ejecución del ciclo.

18. Una expresión es una referencia a una *variable* seguida por un símbolo de asignación y de una *expresión*, o solo una expresión simple. La asignación tiene la semántica de almacenamiento habitual; se encuentra la localidad de la varibale representada por *variable*, luego se evalúa la subexpresión a la derecha de la asignación, se verifican los tipos de variable y de la expresión, y se almacena el valor de la subexpresón en la localidad dada. Una *variable* puede ser simple o un arreglo. Un subíndice negativo provoca que el programa se detenga. Sin embargo, no se verifican los límites superiores de los subíndices.

19. Una *expresión simple* se compone de operadores relaciones que no se asocian o de operadores lógicos que tampoco se asocian. El valor de una *expresión simple* es el valor de su *operando* si no contiene operadores relaciones o lógicos, o bien, 1 si el operador relacional se evalúa como verdadero, cero si se evalúa como falso.

20. Los *términos* y *operandos* representan la asociatividad y precedencia típicas de los operadores aritméticos. En la división entera el resultado se trunca.

21. Un *factor* es una *expresión* encerrada entre paréntesis, una *variable*, una *llamada a una función*, un *entero*, un *flotante*, un *doble*, un *caracter*, o una *cadena*.

    Algunos de estos valores los calcula el analizador léxico. Una variable de arreglo debe estar subindizada, excepto en el caso de un *bloque de sentencias* por un ID simple y empleada en una llamada de función con un parámetro de arreglo.

22. Se debe recordar que se pueden hacer operaciones aritméticas entre enteros, enteros y flotantes, enteros y dobles, flotantes y dobles, caracteres y enteros, realizando el casteo hacia el tipo de mayor longitud. Un char solo se puede castear a int, un int a float y double, un float a double. Los demás tipos de casteo estan prohibidos.

23. Se debe manejar una *Tabla de Tipos*, en ella se insertaran los tipos básicos y a partir de estos se irán construyendo los demás tipos que se pueden definir mediante los arreglos y los tipos compuestos por las estructuras. La *Tabla de Tipos* almacena la siguiente información: {No de tipo, Nombre del Tipo, Tipo Base, Tamaño }

3. Código Intermedio

El código intermedio que genera el compilador es código de tres direcciones. En el código de tres direcciones, hay máximo un operador en el lado derecho de una instrucción, no se permite expresiones aritméticas acumuladas.

Se presenta a continuación una lista de las formas de instrucciones de tres direcciones que debe generar el compilador.

3.0.1. Instrucción de asignación

$x = y\ op\ z$, en donde op es una operación aritmética o lógica binaria, x,y y z son direcciones.

$x = op\ y$, donde op es una operación unaria. En esencia, las operaciones unarias incluyen la resta unaria, la negación lógica, los operadores de desplazamiento y los operadores de cast.

3.0.2. Instrucciones de copia

$x = y$, en dond x se le asigna el valor de y.

3.0.3. Saltos

goto L. La instrucción de tres direcciones con la etiqueta L es la siguienta que se va a ejecutar.

if x goto L e *ifFalse x goto L*. Estas instruacciones se ejecutan a continuación de la instrucción con la etiqueta L si x es verdadera y falsa, respectivamente. En cualquier otro caso, la siguiente instrucción en ejecutarse es la instrucción de tres direcciones que siga en la secuencia.

if x relop y goto L, es un salto condicional que aplica un operador relacional a x y y y ejecuta a continuación la instrucción con la etiqueta L si x predomina en la relación *oprel* con y. Si no es así, se ejecuta a continuación la siguiente instrucción de tres direcciones que vaya despúes de *if x relop y goto L*.

3.0.4. Llamadas a procedimientos

Las llamadas a los procedimientos y los retornos se implementan mediante el uso de las siguientes instrucciones: *param x* para los parámetros; *call p, n* y *y= call p,n* para las llamadas a procedimientos y funciones, respectivamente; y *return y*, en donde y representa a un valor de retorno.

3.0.5. Copia Indexada

x[i] = y y *x = y[i]*. La instrucción *x = y[i]* establece el valor de x al valor en la ubicación que se encuentra a i unidades de memoria más allá de y. La instrucción *x[i]= y* establece el contenido de la ubicación que se encuentra a i unidades más allá de x, con el valor y

4. Arquitectura Destino

La computadora destino modela una máquina de tres direcciones con operaciones de carga y almacenamiento, operacaciones de cálculo , operaciones de salto y saltos condicionales. La computadora subyacente es una máquina con direccionamiento por bytes y n registros de propósito general: R0, R1 , ... Rn-1. Un lenguaje ensamblador completo tendría muchas instrucciones; pero solo se va a utilizar un conjunto bastante limitado de instrucciones, y se da por hecho que para cada tipo de dato se tiene una instrucción para realizar sus operaciones, ejemplo la suma de enteros ADD, la suma de flotantes ADDF. La mayoría de instrucciones consiste en un operador, seguido de un destino, seguido de una lista de operandos de origen. Puede ir una etiqueta después de una instrucción.

El conjunto de instrucciones de la máquina permite realizar todas las operaciones descritas en el párrafo anterior y se presenta a continuación la forma en que están agrupadas para su mejor comprensión.

4.0.6. Operación de Carga

Operaciones de carga: la instrucción *LD destino, direccion* carga el valor que hay en dirección y la carga en destino. Esta instrucción denota la asignación *destino = direccion*. La forma más común de esta instruccción es *LD r, x* que carga el valor que hay en la ubicación x y lo coloca en el registro r. Una instrucción de la forma *LD r1, r2* es una copia de un registro a otro, en la cual se copia el contenido del registro r2 al registro r1.

4.0.7. Operación de Almacenamiento

Operaciones de almacenamiento: La instrucción *ST x, r* almacena el valor que hay en el registro r y lo coloca en la ubicación x. Esta instrucción denota la asignación *x= r*

4.0.8. Operaciones de Cálculo

Operaciones de cálculo de la forma *OP destino, orig1, org2*, en donde OP es un operador como ADD o SUB, y destino, org1, org2 son ubicaciones, no necesariamente distintas. El efecto de esta instrucción de máquina es aplicar la operación representada por OP a los valores en las ubicaciones org1 y org2, y colocar el resultado de esta operación en la ubicación destino. Por ejemplo, SUB r1, r2, r3 calcula r1 =r2-r3. Cualquier valor que haya estado almacenado en r1 se pierde, pero si r1 es r2 o r3, el valor anterior se lee primero. Los operadores unarios que sólo reciben un operando no tienen org2.

4.0.9. Operaciones de Salto

Saltos incondicionales. La instrucción *BR L* provoca que el control se bifurque hacia la instrucción de máquina con la etiqueta L.

Saltos condicionales de la forma *B condicion, L*, en donde *L* es una etiqueta y *condicion* representa la dirección donde se colocó el resultado de la evaluación.

4.1. Memoria

La arquitectura destino cuenta con 6k de memoria en RAM. El tamaño de palabra es de 2 bytes, es decir; 16 bits.

La memoria esta organizada de la siguiente manera:

1. Un área de código, la cual contiene el código destino ejecutable. El tamaño del código destino se puede terminar en tiempo de compilación. Comienza en la dirección 0 hasta la dirección 2047

2. Un área de datos, para contener constantes globales y demás datos que genera el compilador. El tamaño de las constantes globales y de los datos del compilador se determinan en tiempo de compilación. La dirección de inicio de este bloque es 2048 y termina en 3071

3. Un área de Montículo, para contener objetos de datos que se asignan y se liberan durante la ejecución del programa. El tamaño del montículo no puede determinarse en tiempo de compilación. Comienza en la direccion 3072 y termina en 4095. Las direcciones del montículo se controlan de forma indexada con un apuntador HP (Heap Pointer)

4. Un área de Pila administrada en forma dinámica, para contener los registros de activación a medida que se crean y se destruyen, durante las llamadas a los procedimientos y sus retornos. Al igual que el Montículo, el tamaño de la Pila no se puede determinar en tiempo de compilación. Las direcciones disponibles para la pila comienzan en 6143 y terminan en 4096. Se dispone de un registro apuntador a la pila que se conoce como SP (Stack Pointer)

4.2. Modos de direccionamiento

La máquina destino puede realizar diferentes modos de direccionamiento y a continuación se prensetan todos:

- En las instrucciones, una ubicación puede ser el nombre de una variable x que haga referencia a la ubicación de memoria que esta reservada para x.

- Una ubicación también puede ser una dirección indexada de la forma a(r), en donde a es una variable y r un registro. La ubicación de memoria denotada por a(r) se calcula tomando el valor de a y sumándolo al valor en el resgitro r. Por ejemplo, la instrucción *LD R1, a(R2)* tiene como efecto de establecer R1 = contenido(a + R2), en donde contenido(x) denota el contenido del registro o la ubicación de memoria representada por x. Este modo de direccionamiento es útil para acceder a los arreglos, en donde a es la direción base del arreglo (es decir, la dirección del primer elemento) y r contiene el número de bytes después de esa dirección que deseamos avanzar para llegar a uno de los elementos del arreglo a.

- Permite por último el direccionamiento constante inmediato. La constante lleva el prefijo #. La instrucción LD R1, #100 carga el entero 100 en el registro R1, y ADD R1, R1, #100 carga el entero 100 en EL registro R1

Ejemplos:

LD R0, R1	copia el contenido del registro R1 al registro R0
LD RO, M	carga el contenido de la ubicación de memoria M al registro RO
LD R1	

4.3. Asignación de pila

En la asignación de pila la posición de un registro de activación para un procedimiento no se conoce sino hasta el tiempo de ejecución. Por lo general, esta posición se almacena en un registro, por lo que se puede acceder a las palabras en el registro de activación como desplazamiento a partir del valor en ese registro. El modo de direccionamiento indexado para nuestra máquina destino es conveniente para este fin.

Las direcciones relativas en un registro de activación pueden tomarse como desplazamientos a partir de cualquier posición conocida en el registro de activación.

Solo se pueden usar desplazamientos positivos, mantienendo en el resgistro SP un apuntador al inicio del registro de activación en la parte superior de la pila. Cuando ocurre una llamada a un procedimiento, el procedimiento que llama incrementa a SP y transfiere el control al procedimiento que lo llamó. Una vez que el control regresa al emisor, decrementa a SP, con lo cual se desasigna el registro de activación del procedimiento al que se llamó.