CIENCIA DE LA COMPUTACION

COMPILADORES

MANUAL DE REFERENCIA – INFORME FINAL

LENGUAJE PROPUESTO: GRAF

ALUMNO:
Alexander Pinto De la Gala

julio de 2020

“El alumno declara haber realizado el presente trabajo de acuerdo a las normas de la Universidad Católica San Pablo”

_____

MANUAL DE REFERENCIA

LENGUAJE GRAF

1. Introducción

Este manual de referencia describe el lenguaje de programación Graf. El compilador ha sido desarrollado en Lenguaje Python 3.7 y tiene como lenguaje objetivo C++ std11.

1.1 Notación

Las descripciones de análisis léxico y sintáctico utilizan una notación similar a BNF:

name: *lc_letter (lc_letter | "_")**

lc_letter: *"a"…"z"*

La primera línea indica que un name (nombre) inicia con una letra y puede seguir con n letras con sub-guiones.

Una lc_letter (letra) puedes ser un simple carácter de la 'a' a la 'z'.

Los operadores :

- Asterisco * : indica 0 a más
- Suma + : indica uno o más
- *[phrase]* : phrase es opcional
- <…> : información adicional

2. Análisis Léxico

2.1 Estructura de línea

2.1.1 Líneas lógicas

El final de una línea lógica es representado por el salto de línea.

2.1.2 Líneas físicas

El final de una línea física está dado por el salto de línea o caracter *(\n)*.

2.1.3 Unión de líneas

Esta versión no incluye esta característica.

2.1.4 Agrupación de sentencias

Las sentencias son agrupadas con el uso de los caracteres *'{'* para iniciar el agrupamiento y *'}'* para terminar el agrupamiento. Se utiliza los mismos tokens para estos caracteres.

2.1.5 Espacio en blanco entre tokens

Los espacios en blanco son utilizados para separar tokens, de manera que no exista confusión entre ellos.

2.2 Identificadores y keywords.

identifier: *(letter|"_")(letter|digit|"_")**

letter: *lowercase | uppercase*

lowercase: *"a" … "z"*

uppercase: *"A"…"Z"*

digit: *"0"…"9"*

Identificadores son ilimitados en longitud. Case sensitive.

2.2.1 Keywords

Los siguientes son palabras reservadas por el lenguaje y no puede ser usados en combinación con identificadores ordinarios:

if *then* *for* *in* *node* *edge*

2.3 Literales

Literales son notaciones para valores constantes de tipos incluidos.

2.3.1 Literales numéricos

En esta versión hay un literal numérico: plain integer.

integer *: nonzerodigit digit* | "0"*

nonzerodigit *: "1"…"9"*

2.3.2 Literales de objetos

Se incluyen dos literales correspondiente a la abstracción de elementos básicos de una estructura de datos e tipo grafo.

node : *"node " identifier*

El literal *node* construye un nodo de un grafo en donde su atributo único es del tipo de literal elegido.

edge: "edge" *identifier*

El literal *edge* construye una arista de un grafo en donde su atributo único es del tipo de literal elegido.

2.4 Operadores

Los siguientes tokens son operadores:

+ *-* *** *==*

3. Expresiones

En adelante la notación BNF extendida será usada para describir la sintaxis, no el análisis léxico.

3.1 Átomos

Átomos son los elementos más básicos de las expresiones. Los átomos más simples son identificadores o literales. Pueden estar encerrados en paréntesis o corchetes.

átomo: identificador | literal

3.1.1 Identificador (Nombres)

Un identificador ocurre cuando un átomo hace referencia al nombre de un binding local o global.

Cuando el nombre es ligado a un objeto, la evaluación de ese átomo conduce hacia el objeto.

3.1.2 Literal

Graf soporta literal de cadena y numérico entero.

literal: stringliteral | integer

La evaluación de un literal conduce a un objeto del tipo dado (*string, integer)* con el valor dado.

Todos los literales corresponden a tipos de datos inmutables, y por lo tanto la identidad del objeto es menos importante que su valor.

3.2 Operadores aritméticos binarios.

Las operaciones aritméticas binarias tienen niveles convencionales de prioridad.

Existen dos niveles de prioridad:

m_expr: u_expr | m_expr "" u_expr | m_expr "/" u_expr*

a_expr: m_expr | a_expr "+" m_expr | a_expr "-" m_expr

El operador * de multiplicación produce el producto de sus argumentos. Ambos argumentos deben ser números.

El operador / de división produce el cociente de sus argumentos. El resultado de la división tendrá la función "*floor*" debido al tipo *integer* soportado. Ambos argumentos deben ser números.

El operador *+* de adición produce la suma de sus argumentos. Ambos argumentos deben ser números.

El operador – de sustracción produce la resta de sus argumentos. Ambos argumentos deben ser números.

3.3 Operadores de Objetos

Los operadores de objetos permiten operaciones con la abstracción de la estructura de grafos.

asignación: *nodo1 "->" edge*

La arista *edge* es asignada a un nodo 1

Un nodo puede tener asignados varios *edges*.

acceso a edges: *target "=" "!"ident_nodo*

4. Sentencias simples

4.1 Sentencias de asignación

Las sentencias de asignación son usadas para (re)ligar nombres a valores.

assignment_stmt : (target "=")+ target

5. Sentencias compuestas

Las sentencias compuestas contienen grupos de otras sentencias, estas afectan o controlan la ejecución de las otras sentencias de alguna manera. En general las sentencias compuestas contienen múltiples líneas.

5.1 La sentencia *if*

La sentencia *if* es usada como condicional de ejecución:

if_stmt: *"if" expresión "{" suite "}"*

*("else if" expression "{" suite "}")**

["else" "{" suite "}"

Selecciona exactamente una de las *suites* para evaluar las expresiones una por una hasta que una resulte verdadera entonces esa *suite* es ejecutada (y ninguna otra parte de la sentencia *if* es ejecutada).

Si todas las expresiones son falsas la *suite* de la sentencia *else* ejecutada si está presente.

"{" suite "}"

La *expression list* es evaluda una vez, debería representar una lista. La *suite* entonces es ejecutada una vez por cada item en la secuencia, en el orden de índices ascendientes.

ARQUITECTURA DEL COMPILADOR

1. GRAMÁTICA

El compilador tiene como base una gramática libre de contexto de tipo LL1.

Program := Data_decs Block

Data_decs := Data_dec DataP

DataP := lambda | Data_decs

Data_dec := int Var semicolon | node Var semicolon | edge Var semicolon

Block := abrellave Statements cierrallave semicolon

Statements := Statement StP

StP := lambda | Statements

Statement := Assignment

Assignment := Var AssP

AssP := equal Expression semicolon | flecha Var semicolon

Assignment := if Expression semicolon Block

Expression := Term ExpressionP

ExpressionP := Add Term ExpressionP | lambda

ExpressionP := Rest Term ExpressionP | lambda

Term := Factor TermP

TermP := Mult Factor TermP | lambda

Factor := AbreParentesis PosDBI | Num | Var

PosDBI := AbreParentesis ConfDBI | ConfDBI

ConfDBI := Expression CierraParentesis DBD

DBD := lambda | CierraParentesis

La gramática ha tenido que ser adaptada para evitar recursión por la izquierda, por lo que se ha debido factorizar cuando las producciones contenían términos repetidos, además de agregar no-terminales de ser necesario.

La gramática luego es procesada de manera que es posible crear una Tabla Sintáctica de Predicción. Para esto debemos identificar los Primeros y los Siguientes de acuerdo con reglas apropiadas para una gramática LL1.

2. ANALIZADOR LÉXICO

El Analizador Léxico contiene un diccionario que permitirán construir los tokens desde nuestro archivo de entrada.

Está conformado básicamente por tres funciones que se encargan de reconocer si el token es un número, una variable o un terminal. Cada token es un array que lleva información como el identificador utilizado, el valor asociado, el número de línea y número de columna donde se halla en el archivo de entrada. Cada final de línea también es considerado un token.

En este módulo se integra recuperación de errores a nivel léxico.

Los tokens son agrupados en una lista la cual es enviada al Analizador Sintáctico.

3. ANALIZADOR SINTÁCTICO

El analizador sintáctico recibe la lista de tokens desde al analizador léxico.

Para el análisis utiliza una pila de tokens y la lista de tokens la cual tendrá una función de tipo cola.

La cabecera de la pila y la cola son comparadas en caso de match los tokens son retirados. Si no fuera así se va creando el Árbol Sintáctico, para ellos se tienen una clase Nodo y Árbol los cuales recibirán estos valores y tienen las funciones necesarias para inserción de nodos y su impresión a manera de debug.

En caso de existir un error la ejecución se detendrá. Esto debe ser mejorado en futuras versiones.

El resultado final será un Árbol Sintáctico el cual generará un Árbol Semántico que forma parte del Analizador Semántico.

4. ANALIZADOR SEMÁNTICO Y GENERACIÓN DE CÓDIGO

El árbol sintáctico genera un árbol semántico por medio de un patrón interprete. Esta acción se realiza clase a clase. Para esto requerimos de una función que reconozca clases de una lista de clases.

Cada una de estas clases tienen un conjunto de producciones apropiadas que apuntan a otras clases.

El análisis semántico se realiza llamando a interpretación del nodo raíz el cual a su vez hará el llamado a los nodos hijos de manera recursiva. Las hojas finalmente devolverán la interpretación adecuada a sus respectivos padres y finalmente al nodo raíz.

En esta etapa se implementa dos recuperaciones de errores por medio de gramática. Los dos están referidos al uso no balanceado de paréntesis.

Finalmente, la se obtiene el código generado hacia el lenguaje objetivo C++.

5. MODELO DE ENTRADA Y SALIDA

A continuación, se muestra un ejemplo de código de entrada y salida:

int a

int b

int c

node N

node M

edge S

{b=((100005+3)

a=2

M=3

c = (10 + 5))

N -> S

}

```
      #include <iostream>
  #include <vector>
 using namespace std;
 struct edge;
 struct node{
  int data;
  vector<edge> vec_edges;
  };
 struct edge{
 vector<node> vec_nodes;
 };
int a;
int b;
int c;
node N;
node M;
edge S;
void main()
{b=100005+3;
a=2;
M.data = 3;
c=10+5;
N.vec_edges.push_back(S);
S.vec_nodes.push_back(N);
}
```