Universidad Nacional
Autónoma de México

Facultad de Ciencias

Compiladores Proyecto Final
Construcción de un Compilador

Profesor:
Adrián Ulises Mercado
Martínez

Ayudantes:
Karla Adriana Esquivel
Guzmán
Carlos Gerardo Acosta
Hernández

1. Proyecto Final

1. Objetivo

   Construir un compilador para el lenguaje definido por la gramática de la sección 2, utilizando lex y yacc, que genere código objeto para MIPS.

2. Gramática.

   programa → declaraciones funciones

   declaraciones → tipo lista ;|ϵ

   tipo → int | float | double | chart | void | struct {*declaraciones*}

   lista → lista, id arreglo | id arreglo.

   arreglo → [numero] arreglo |ϵ

   funciones → func tipo id (argumentos) {*declaracione sentencias*} funciones |ϵ

   argumentos →¿lista_ argumentos|ϵ

   lista_argumentos → lista_argumentos, tipo, id parte_arreglo | tipo id part_arreglo.

   parte_arreglo → [] parte_arreglo |ϵ

   sentencia → sentencias sentencias | if (condición) sentencias | if (condición) sentencias else sentencias | while(condición) sentencias | do sentencias while(condición); | for (sentencia; condición; sentencias) sentencia | parte_izquierda = expresión; | return expresión; | return; | {*sentencia*} | switch {*sentencia*} { caso predeterminado } | break;| print expresión

   casos → case: numero sentencia predeterminado |ϵ

   predeterminado → default: sentencia |ϵ

   parte_izquierda → id | var_arreglo | id.id

   var_arreglo → id[expresion] | var_arreglo [expresión]

expresión → expresión + expresión | expresión - expresión | expresión * expresión | expresión / expresión | expresión % expresión | var_arreglo | cadena | numero | caracter | id (parámetros)

parámetros → ϵ | lista_parametros

lista_parametros → lista_parametros, expresión | expresión

condición → condición || condición | condición & condición | !condición | (condición) | expresión relacional expresión | true| false

relacional → < | > | >= | <= | != |==

3. Análisis léxico.

Objetivo:

Elaborar el analizador léxico para la gramática de la sección 2, utilizando lex.

Investigar:

1. Uso de estados léxicos en lex

La herramienta Lex tiene una manera de reconocer expresiones regulares cuando se cumple una condición. Dichas expresiones regulares se pueden guardar en un estado, con la sintáxis $< nombre/estado >$, tal que se activara cuando el analisador léxico se encuentre en este estado. Un ejemplo sería:

<Dígito> [⋀”] { printf (Encontraste un número) }

Las expresiones regulares pueden estar en varios estados y dichos estados deben de estar declarados en el encabezado de tu archvo lex.

2. Leer la entrada desde un archivo usando yyin

Por defecto, el analizador léxico generado por lex lee caracteres de la entrada esándar stdin. Si se desea cambiar este comportamiento basta con hacer que la variable yyin (que es de tipo FILE *) apunte a otro fichero (previamente abierto para lectura con fopen) antes de llamar a yylex, lo cual nos permite leer datos de entrada de un archivo de texto, si deseas escribir un archivo sólo hace falta usar la función yyout.

Descripción:

a) int, float, double, char y void: son palabras reservadas para declarar los tipos de las variables, que se deben sustituir por otras en algún otro idioma.

b) struct es una palabra reservada para definir variables del tipo registro o estructura, se debe sustituir por una palabra en otro idioma.

c) numero: debe poder reconocer tres tipos de números (enteros, flotantes y de doble precisión) relacionados con los tipos

d) id: son los identificadores para el lenguaje de programación.

e) if, else, while, do, return, switch, break, return, print, case, default, true y false: se deben sustituir por palabras reservadas den otro idioma que no sea el inglés, mismo idioma usado para las palabras reservadas de 1 y 2.

f) cadena: son secuencias de caracteres.

g) caracter: debe reconocer solo un caracteres.

h) los operadores: ||, & & y !: deben ser sustituidos por algún otro símbolo o palabras.

4. Consideraciones.

a) Constantes de caracteres.

Este lenguaje admite dos tipos de constantes de cadenas: cadenas y de caracteres. Una cadena es cualquier secuencia de caracteres tipo ASCII. El símbolo sirve para usar caracteres especiales. Los caracteres son cualquier símbolo ASCII.

b) Comentarios.

Este lenguaje de programación acepta comentarios que comienzan con ¡* y terminan con *¿, se deben implementar utilizando estados en lex, un error léxico debe marcarse si el comentario no se cierra.

c) Espacios en blanco.

Todos los espacios en blanco deben ser reconocidos para poder ser ignorados al igual que los comentarios.

d) Palabras reservadas.

Las palabras reservadas para el lenguaje se deben definir en cualquier otro idioma que no sea el inglés.

e) Identificadores.

Los identificadores son nombres de las variables, funciones y estructuras.

5. Documentos a entregar:.

6. Descripción del problema.

Se desea construir un pequeño compilador que reconozca la gramática de la sección dos, utilizando la herramienta lex y yacc en c, y genere código objeto en MIPS (Microprocessor without Interlocked Pipeline Stages) un programa que reconoce código máquina para nuestro programa pueda ser ejecutado a un bajo nivel por la computadora. En esta primera sección únicamente desarrollaremos el análisis léxico de dicha grmática que consiste en buscar los componentes léxicos o palabras que componen el programa fuente, según las reglas gramaticales. La entrada del analizador léxico podemos definirla como una secuencia de caracteres. El analizador léxico tiene que dividir la secuencia de caracteres en palabras con significado propio y después convertirlo a una secuencia de terminales desde el punto de vista del analizador sintáctico, que es la entrada del analizador sintáctico.

En otras palabras sólo realizaremos el reconocimiento de tokens dado un archivo, teniendo en cuenta que los espacios en blanco no cuentan como token y son ignorados de igual manera que los comentarios, así que sólo crearemos estos lexemas con respecto a las expresiones regulares, más adelante realizaremos el árbol de derivación de cada expresión para su anális sintáctico y semántico.

7. Análisis del problema.

a) Expresiones regulares de los componentes léxicos.

Palabras Reservadas:
si|$contrario$|$regresa$|$imprime$|haz|$mientras$|$caso$|$otro$|$verdadero$|$falso$

Tipos de datos:
$registro|entero|doble|caracter|vacio$
Operaciones booleanas:
$registro|entero|doble|caracter|vacio$
Operacion de asignación:
$==$
Operaciones de orden:
$<= \mid > \mid <= \mid <$
Operaciones aritmeticas:
$+ \mid - \mid * \mid / \mid \%$
Simbolos especiales:
$(\mid) \mid \{ \mid \} \mid [\mid] \mid .$
Identificadores:
$[a-zA-Z]^*[a-zA-Z0-9]^*$
Numeros Enteros:
$[-+]?[0-9]^+$
Numeros Dobles:
$[-+]?[0-9]*[.][0-9]^+$
Numeros Flotantes:
$[-+]?[0-9]*[.][0-9]^+[Ff]$
Caracteres:
$'[.]'$
Cadenas:
"[.]*"

b) Consideraciones y restricciones.

8. Diseño e implementación.

   a) De ser posible incluir el autómata que reconocerá al lenguaje.

   b) La organización del código fuente.

   Se tiene un archivo .flex así como uno con las macros que definen los tipos de tokens que tiene nuestro lenguaje de programación

9. Resultados y conclusiones.

   El código fuente:

   a) Cada función debe contener una breve descripción de que es lo que hace y quién la programó.

b) Se debe respetar la identación de dependencia de instrucciones.

c) Se debe usar.

Conclusiones:

El analizador léxico es la primera fase de un compilador. Su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis.

Como el analizador léxico es la parte del compilador que lee el texto fuente, también puede realizar ciertas funciones secundarias en la interfaz del usuario, como eliminar del programa fuente comentarios y espacios en blanco en forma de caracteres de espacio en blanco, caracteres TAB y de línea nueva. Otra función es relacionar los mensajes de error del compilador con el programa fuente.

En algunas ocasiones, los analizadores léxicos se dividen en una cascada de dos fases; la primera, llamada .^{ex}amen", y la segunda, .^{a}nálisis léxico". El examinador se encarga de realizar tares sencillas, mientras que el analizador léxico es el que realiza las operaciones complejas.

Las conclusiones principales sobre la primer parte de la creación del compilador son: primero que nada debes generar las expresiones regulares de la grmática ya que estas expresiones nos ayudarán a saber que palabras son aceptadas en nuestro lenguaje y segundo la división en lexemas o tokens de una cadena debe tener en cuenta aquellas palabras terminales, lexemas que serán ignorados y por ello al leer se no se tomarán en cuenta y las concideraciones particulares de tu leenguaje, es decir si aceptará símbolos especiales o caracteres especiales, las palabras reservadas de tu lenguaje y la sintaxis para la creación de comentarios.

Es esencial modelar de forma correcta el problema a resolver, ya que hay muchas cosas que se pueden incluir dentro del analisis léxico que bien podrian formar parte de otra fase de compilación. El enfoque que nosotros tomamos fue hacer el analisis léxico más simple y funcional para poder modificarlo conforme vayan surguiendo nuevos desafios.

d) **Análisis Sintáctico**

1) Investigar:
Para resolver la ambigüedad del if-else, es que se debe asignar una precedencia a else como si fuera el operador de mayor precedencia.(%prec).

La gramática if-else es la siguiente:

S → if COND then S
S → if COND then S else S
S → id = EXPR

en esta gramatica podemos notar que se encunetran conflictos desplazar/reducir ante una entrada como:

$$\text{if } COND_1 \text{ then if } COND_2 \text{ then } S_1 \text{ else } S_2$$

Cuando un compilador de C encuentra una sentencia como esta opta por construir el segundo arbol sintactico siguiendo la regla general ëmpareja cada else con el then sin emparejar anterior mas cercano". Esta regla para eliminar ambiguedades se puede incorporar directamente a la gramatica añadiendo algunas reglas de produccion adicionales:

S → SCompleta | SIncompleta

SCompleta → if COND then SCompleta else SCompleta | id =EXPR

SIncompleta → if COND then S | if COND then SCompleta else SIncompleta

que asocian siempre el else al if más inmediato. De forma que una SCompoleta es o una proposicion if-else-then que no contega proposiciones incompletas o cualquier otra clase de proposicion no condicional (como ejemplo una asignacion).Es decir dentro de un if con else solo se permiten if completos, lo que hace que todos los else se agrupen antes de empezar a reconocer los if sin else.

2) Descripción del problema a resolver (no del progamar)

Como mencionamos anteriormente el objetivo de este proyecto es crear un compilador que reconozca la gramática de la sección 2, en la parte anterior generamos un programa que dividia en tocken o lexemas el código a analizar. En esta sección se creará un programa para hacer el análisis sintáctico del compilador que recibe los tockens de la etapa de análisis léxico y construye un árbol de análisis sintáctico o árbol de parseo que nos sirve para determinar si una expresión es

generada con una gramática dada.

Algunas concideraciones importantes son que la gramática que estes conciderando debe ser libre de ambigüedad y quitar la recursividad izquierda o recursividad inmediata izquierda ya que al crear la tabla de símbolos genera problemas si no cumple estas propiedades, es decir no puedes asegurar si una expresión proviene de una gramática dada si no son libres de ambigüedad y tienen recursividad izquierda o derecha.

3) Análisis del problema

   a' Eliminar la ambigüedad del problema

   b' Consideraciones y restricciones.

4) Diseño e implementación

   a' Representarla en notación EBNF.

   b' Diagramas de sintaxis del lenguaje

   c' La organización del código fuente.

5) Resultados y conclusiones individuales.

Referencias

[1] https://stackoverflow.com/questions/8859710/lex-how-to-run-compile-a-lex-program-on-commandline

[2] Compiladores principios, técnicas y herramientas, Alfred V Aho, segunda edición.