Arquitectura de compilador LPNCC

Compilador de un Sublenguaje de C

Proyecto compilers2019_2

Versión

Fecha	Versión	Autor	Organización	Descripción
02/03/19	**1.0**	**Oscar Gutierrez Castillo Francisco Javier Espinoza Jimenez**	La Peste Negra (LPN)	Creación de Documento

Definición de la arquitectura del compilador

La implementación del compilador se basa en el desarrollo tradicional de un compilador, es decir, análisis léxico, análisis sintáctico y finalmente la generación de código. Teniendo como soporte una tabla de símbolos y detectores de errores que nos permitirán generar y validar la información que poseemos.

El desarrollo del compilador será en Elixir, por lo que las implementaciones caerán directamente sobre módulos independientes en su funcionamiento interno, dependiendo exclusivamente de la salida directamente anterior del módulo al cual van conectados en el diagrama descrito en la imagen 1.

Creación de diagramas describiendo los módulos del compilador

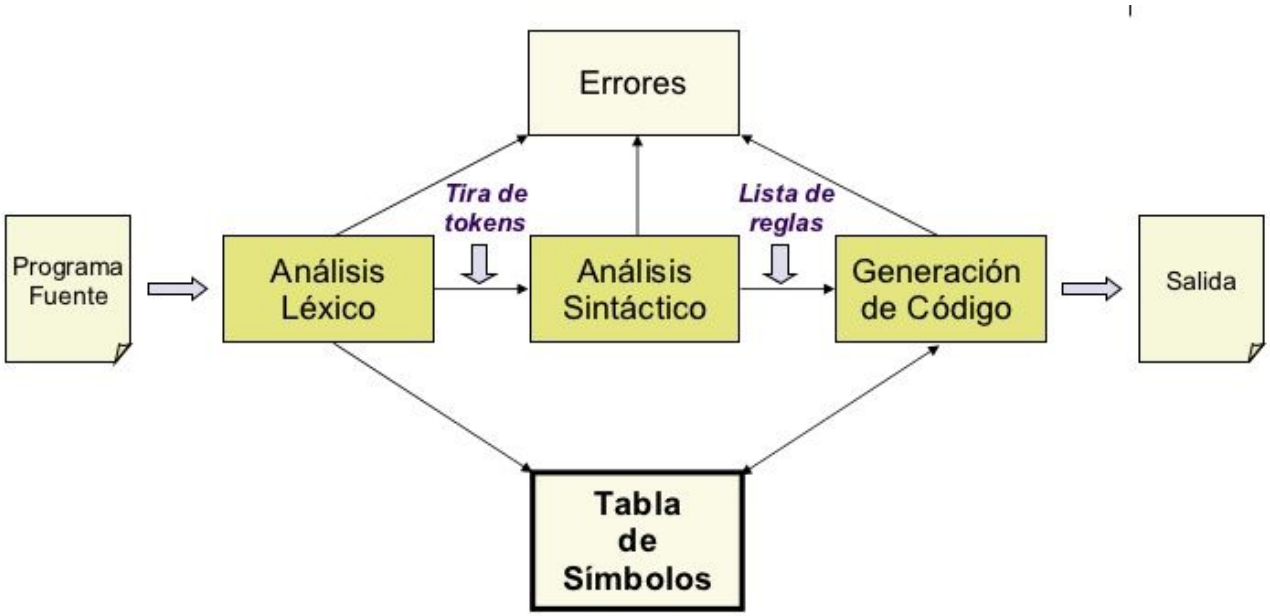

Imagen 1: Esquema básico de un compilador

1. Módulo de análisis léxico
    a. Elementos insumo:
        i. Programa fuente
        ii. Tabla de símbolos

    b. Tarea designada:
        i. Deteccion y clasificacion de los símbolos que posee el archivo fuente.
        ii. Primero el módulo debe proceder a limpiar de caracteres no deseados como saltos de línea, tabulaciones etc y dejar simplemente espacios entre los símbolos.
        iii. Posterior a la limpieza debemos de clasificar respecto a la lista de símbolos que poseemos los caracteres del archivo fuente
        iv. Es importante dejar claro que en esta etapa el compilador NO realiza ningún tipo de validación respecto a el orden o sintaxis de los símbolos.
    c. Salida:
        i. Lista de tokens
2. Módulo de análisis de sintáctico
    a. Elementos insumo:
        i. Lista de tokens
    b. Tarea designada:
        i.
    c. Salida:
        i. Árbol AST
3. Módulo de generación de código
    a. Elementos insumo:
        i. Árbol AST
        ii. Tabla de símbolos
    b. Tarea designada:
    c. Salida:
        i. Código ensamblador para arquitectura x86_64
4. Tabla de símbolos
5. Manejo de errores

Definición de estándares de codificación

Disposición del Código Fuente (Layout)

- Uso dos espacios por nivel de indentación. No utilizar tabulaciones.

```
defmodule Lexer do
__def scan_words(words) do
____Enum.flat_map(words, &lex_raw_tokens/1)
__end
...
```

- Uso los finales de línea de Unix

NOTA:Los usuarios de *BSD/Solaris/Linux/OSX ya están cubiertos por defecto, pero los usuarios de Windows tendrán que prestar especial atención.

- Uso de espacios alrededor de operadores, después de comas, dos puntos y de punto y coma. No colocar espacios alrededor de parejas como llaves, parentesis, etc.

```
...
"{" <> rest ->
        {:open_brace, rest}
...
```

- Utilizar líneas en blanco entre `def`s para separar las funciones en párrafos lógicos, pero no dejar líneas en blanco cuando tengas `def`s de una sola línea cuyos argumentos representan pattern matching.

Sintaxis

- Usa paréntesis cuando `def` tenga argumentos, y omítirlos cuando no.

```
# recomendado
def some_function(arg1, arg2) do
  # body omitted
end
```

```
def some_function do
  # body omitted
end
```

- Añadir una línea en blanco tras una "asignación" multilínea como una pista visual de que ha terminado.
- No utilizar `do:` para `if/unless` multilínea

```
# recomendado
if success? do
  IO.puts 'success'
else
  IO.puts 'failure'
end
```

Nombrado

- Uso de `snake_case` para atoms, funciones y variables.

```
# recomendado
:some_atom
some_var = 5
def some_function do
  ...
```

```
end
```

- Uso de `CamelCase` para módulos (mantén los acrónimos como HTTP, RFC, XML en mayúsculas).

```
# recomendado
defmodule SomeModule do
  ...
end
defmodule SomeXML do
  ...
end
```

Comentarios

- Escribir código expresivo e intenta transmitir la intención de tu programa a través de flujos de control, estructura y nombrado.

- Los comentarios que sean más largos de una palabra se escribirán capitalizados, y las frases utilizarán signos de puntuación. Usa un espacio tras cada punto.

Comentarios de Anotación

- Las anotaciones se escriben normalmente en la línea inmediatamente superior al código que anotan.
- La palabra clave para la anotación estará completamente en mayúsculas, seguida de dos puntos y un espacio, a continuación se añade la nota que describe el problema.

En la siguiente tabla se muestra el estándar de nombrado para las variables

Convención de los nombres de módulos y funciones

Objeto	Convención de nombre
Módulo	ModuleExample
Funcion	function_example

Ejemplos

Objeto	Convención de nombre
Modulo	Lexer
Modulo	Parser

funcion	lex_raw_tokens
funcion	get_constant

Validar que se corran herramientas de inspección de código

Llevar métricas del código

Atributos de Métrica

Líneas de código	**150 lineas**
Tamaño del programa	**9.29 MB (9,746,049 bytes)**
Índice de mantenibilidad	**45**
Complejidad ciclomática	
Acoplamiento de clases	

Líneas de código: Número de líneas en la implementación del proyecto

Índice de mantenibilidad: Es un valor calculado a partir de otras métricas. Comprende entre 0 y 100 (de 0 a 10 es poco mantenible, de 10 a 20 es moderadamente mantenible y por encima de 20 es mantenible).

Tamaño del programa: Tamaño en bytes que ocupa en memoria el programa.

Complejidad Ciclomática: Es una medida del número de caminos independientes dentro de un fragmento de código.

Acoplamiento de clases: Indica con cuantas otras clases estamos relacionado o conectados.

https://github.com/rrrene/credo