Reporte escrito proyecto final - Compiladores 2022 II

AHUMADA, SANTIAGO[1], MORA, DAVID[2], AND MÉNDEZ, EDUARD'S[3]

[1] *Facultad de ciencias, 2022. Universidad Nacional de Colombia, Bogotá D.C.*

Compiled December 4, 2022

Resumen. En este proyecto hemos desarrollado a baja escala un lenguaje de programación cuyo compilador está construido con `C/C++`. Se han programado todas las fases de un compilador: Un analizador léxico con `Lex`, un parser con `Bison` y un ensamblador con `LLVM`.
Palabras clave. Lex, Gramática, Bison, AST.

https://github.com/eguar11011/ProyectoCompiladores.git

CONTENIDO

1. INTRODUCCIÓN

El problema general que pretendemos abordar con en este proyecto es aplicar los conocimientos aprendidos a lo largo del curso: Las partes de un compilador, el analizador léxico de este, el parser y el assembler.

Con base en lo anterior, el problema específico que atacamos es el desarrollo de un lenguaje de programación sencillo, el cual incorpora la definición de funciones y sus respectivas llamadas, creación de variables a las que se les puede asignar datos, y operaciones matemáticas básicas con enteros y números de punto flotante con doble precisión.

La presentación de este proyecto se encuentra en `https://youtu.be/p5wYOO4d-6A`.

2. MATERIALES Y MÉTODOS

- **Solución computacional**. El lenguaje de programación fue construido con base en los lenguajes C y C++. C se utilizó para el desarrollo del analizador léxico en Flex y para el parser en Bison. C++ Se utilizó para desarrollar código de manera más eficiente con la colección de módulos de compilador reutilizables del proyecto LLVM. En resumen, las componentes del compilador del lenguaje son:
  - Gramática. Las especificaciones de la gramática se especifican en el archivo `parser.y`. Por ejemplo, el manejo gramatical de una expresión es:

```
// parser.y
expr : ident TEQUAL expr { $$ = new
     NAssignment(*$<ident>1, *$3); }
| ident TLPAREN call_args TRPAREN { $$ = new
    NMethodCall(*$1, *$3); delete $3; }
| ident { $<ident>$ = $1; }
| numeric
| expr comparison expr { $$ = new
    NBinaryOperator(*$1, $2, *$3); }
| TLPAREN expr TRPAREN { $$ = $2; }
;
```

Además, según las especificaciones de `Bison` y la manera en la que construimos el proyecto, la gramática es de tipo **LR1**.

  - Analizador léxico. Construido con `Flex`. Como se estudió en el curso, este tendrá como objetivo separar las cadenas de caracteres en tokens. En particular, aprovechamos la experiencia obtenida en el desarrollo del primer proyecto del curso, en donde pudimos crear un lexer basado en expresiones regulares.
  - Analizador sintáctico. Construido con `Bison`.El objetivo principal de este componente es generar árboles de sintaxis abstracta cada vez que se tokeniza una instrucción.
  - Assembly. Construido con la colección de módulos de compilador reutilizables del proyecto `LLVM`. En esta etapa codificamos cada árbol de sintaxis abstracta en código de máquina.

- **Componentes**. La estructura del proyecto está dada en 5 módulos:
  - `Makefile`. Mostramos los resultados en un archivo.
  - `main.cpp`. Módulo principal. Recibe instrucciones del usuario y las envía al compilador para la primera fase: análisis léxico.

- `node.h`. Definición de los nodos para generar los árboles de sintaxis abstracta.
- `parser.y`. Definición de la gramática de nuestro lenguaje. También definimos el tipo de nodo que representan nuestros símbolos no terminales. Los tipos hacen referencia a la declaración `%union` anterior, por ejemplo, cuando llamamos un identificador (definido por el tipo de unión ident) estamos realmente llamando a un (NIdentifier*). A continuación podrá ver algunos ejemplos de la definición de las producciones en la gramática

```
    # parser.y
block : TLBRACE stmts TRBRACE { $$ = $2; }
 | TLBRACE TRBRACE { $$ = new NBlock(); }
 ;

var_decl : ident ident { $$ = new
     NVariableDeclaration(*$1, *$2); }
   | ident ident TEQUAL expr { $$ = new
        NVariableDeclaration(*$1, *$2, $4); }
   ;

extern_decl : TEXTERN ident ident TLPAREN
     func_decl_args TRPAREN
          { $$ = new NExternDeclaration(*$2,
               *$3, *$5); delete $5; }
        ;
```

- `tokens.l`. Especificación de los tokens para que nuestro lexer pueda separar las instrucciones en identificadores. A continuación podrá ver algunos ejemplos de la definición de los tokens

```
    # tokens.l
"."     return TOKEN(TDOT);
","     return TOKEN(TCOMMA);

"+"     return TOKEN(TPLUS);
"-"     return TOKEN(TMINUS);
"*"     return TOKEN(TMUL);
"/"     return TOKEN(TDIV);
```

- **Configuración experimental**. Para inicializar el proyecto en una máquina local siga estos pasos:

1. Clone el repositorio:
   `$ git clone https://github.com/eguar11011/ProyectoCompiladores.git`
2. Ingrese a la carpeta del repositorio:
   `$ cd ProyectoCompiladores`
3. Ingrese al código fuente:
   `$ cd finalproject`

3. RESULTADOS

- Ejemplos representativos. En este programa creamos la función `do_math()` la cual toma un número, lo multiplica por 5 y le suma 3.

```
   # ./parser
extern void printi(int val)

int do_math(int a) {
  int x = a * 5
  return x + 3
}
   echo(do_math(11))
```

Al correr el anterior código, obtenemos:

```
   # Generating code...
Generating code for 18NExternDeclaration
Generating code for 20NFunctionDeclaration
Creating variable declaration int a
Generating code for 20NVariableDeclaration
Creating variable declaration int x
Creating assignment for x
Creating binary operation 276
Creating integer: 5
Creating identifier reference: a
Generating code for 16NReturnStatement
Generating return code for 15NBinaryOperator
Creating binary operation 274
Creating integer: 3
Creating identifier reference: x
Creating block
Creating function: do_math
Generating code for 20NExpressionStatement
Generating code for 11NMethodCall
Creating integer: 11
Creating method call: do_math
Creating method call: echo
Creating block
Code is generated.
; ModuleID = 'main'
source_filename = "main"

@.str = private constant [4 x i8] c"%d\0A\00"

declare i32 @printf(i8*, ...)

define internal void @echo(i64 %toPrint) {
entry:
  %0 = call i32 (i8*, ...) @printf(i8* getelementptr
       inbounds ([4 x i8], [4 x i8]* @.str, i32 0, i32
       0), i64 %toPrint)
  ret void
}

define internal void @main() {
entry:
  %0 = call i64 @do_math(i64 11)
  call void @echo(i64 %0)
  ret void
}

declare void @printi(i64)

define internal i64 @do_math(i64 %a1) {
entry:
  %a = alloca i64, align 8
  store i64 %a1, i64* %a, align 4
  %x = alloca i64, align 8
  %0 = load i64, i64* %a, align 4
  %1 = mul i64 %0, 5
  store i64 %1, i64* %x, align 4
  %2 = load i64, i64* %x, align 4
  %3 = add i64 %2, 3
  ret i64 %3
}
Running code...
58
Code was run
```

- Reportes cuantitativos. Podemos apreciar en el output que se generan las instrucciones de llvm donde se crean las variables, se realizan las asinganción de variables y operadores.

Donde luego podemos ver la representación intermedia en assembly.

4. CONCLUSIONES

Dentro de las discusiones planteadas en el desarrollo de este proyecto pudimos analizar componente por componente cada una de las fases de un compilador y su importancia en el desarrollo de software moderno.

Al ser un proyecto concreto hemos tenido la oportunidad de entender a mayor profundidad la arquitectura de los compiladores y la forma en que están estructurados los módulos para desarrollar el lexer `Flex`, el parser `GNU Bison` y el assebly `LLVM`.

5. REFERENCIAS

1. Aho, A.V. et al. (2015) Compilers: Principles, techniques, amp; tools. Charlesbourg, Quebec: Braille Jymico Inc.

2. MIT Compiler Design course (no date) YouTube. MIT Open Course Ware. Available at: https://www.youtube.com/playlist?list=PL9pnk7GPwphF_PagGGUhpU1VzV7-j56vg (Accessed: October 16, 2022).

3. Lexical analysis with Flex, for Flex 2.6.3 (Accessed: October 29, 2022) Lexical Analysis With Flex, for Flex 2.6.3: Top. Available at: https://www.cs.virginia.edu/ cr4bd/flex-manual/ (Accessed: November 11, 2022).

4. Setunga, S. (2020) Writing a parser-part I: Getting started, Medium. Medium. Available at: https://supunsetunga.medium.com/writing-a-parser-getting-started-44ba70bb6cc9 (Accessed: November 13, 2022).