ITESM Campus Puebla

Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 301)

Compilador

Ituriel Mejía Garita - A01730875

Alejandro Castro Reus - A01731065

José Antonio Bobadilla García - A01734433

9 de junio de 2023

Introducción

Los compiladores son una parte fundamental de la programación moderna. Su función es transformar código que puede entender el ser humano fácilmente a un lenguaje que puede ser ejecutado por la computadora. Por eso mismo han hecho que la programación sea más accesible y que se pueda escribir código más eficientemente.

En esta actividad se realizó un compilador basado en la gramática de clite, una versión sencilla de un lenguaje similar a Java y C. Esta gramática incluye expresiones, asignaciones, bucles y condicionales. Adicionalmente se le agregaron llamadas de funciones y whiles por lo que es una implementación con capacidades cercanas a las que tienen lenguajes de alto nivel.

Funcionamiento

El compilador usa las librerías de ply.lex y pli.yacc para el análisis léxico y el análisis sintáctico respectivamente. En el archivo de compiler.py se definieron las reglas léxicas de identificadores de variables (id), floats (floatlit), ints (intlit), bools (boollit), newline, palabras reservadas (como else, while, return, for, entre otras) y operadores (como “||”, “&&” o “<=”).

```
literals = ['+','-','*','/', '%', '(', ')', '{', '}', '<', '>', '=', ';', ',', '!']
reserved = {
    'else' : 'ELSE',
    'float' : 'FLOAT',
    'if' : 'IF',
    'int' : 'INT',
    'bool' : 'BOOL',
    'float' : 'FLOAT',
    'char' : 'CHAR',
    'return' : 'RETURN',
    'while' : 'WHILE',
    'for': 'FOR',
    'return': 'RETURN',
    'void': 'VOID'
}

tokens = list(reserved.values()) + ['ID', 'INTLIT', 'FLOATLIT', 'BOOLLIT', 'LTE', 'GTE', 'EQ', 'NEQ',
'AND', 'OR']
```

```
t_ignore  = ' \t'

t_LTE = r'<='
t_GTE = r'>='
t_EQ = r'=='
t_NEQ = r'!='
t_AND = r'&&'
t_OR = r'\|\|'

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value,'ID')    # Check for reserved words
    return t

def t_FLOATLIT(t):
    r'((([0-9]+)?\.[0-9]+)|([0-9]+\.))'
    t.value = float(t.value)
    return t


def t_INTLIT(t):
    r'[0-9]+'
    t.value = int(t.value)
    return t

def t_BOOLLIT(t):
    r'[0|1]'
    return t
```

Igualmente se definieron las reglas sintácticas en el mismo documento. La regla sintáctica más importante y de la que surgen todas las demás reglas es la de program, un program está compuesta de 1 o más funciones que están compuestas de un tipo de retorno (FunctionReturnType), un ID, pueden tener params o no y, entre llaves, Declarations, Statements y un ReturnStatement.

```
def p_Program(p):
    '''
    Program : Function Program
            | empty
    '''
    if len(p) > 2:
        p[0] = Program(p[1], p[2])

def p_Function(p):
    '''
    Function : FunctionReturnType ID '(' ')' '{' Declarations Statements ReturnStatement '}'
             | FunctionReturnType ID '(' Params ')' '{' Declarations Statements ReturnStatement '}'
    '''
    if len(p) > 10:
        p[0] = Function(p[1], p[2], p[4], p[7], p[8], p[9])
    else:
        p[0] = Function(p[1], p[2], False, p[6], p[7], p[8])
```

Declarations son un conjunto de declaration, estas son las asignaciones de variables compuestas por un type (puede ser float, int, bool o char) y un id. Statements son un conjunto de statement, bien puede ser asignación de valores a las variables (Assignment), IfStatement, WhileStatement, ForStatement, o bloques de código (Block).

```
def p_Declarations(p):
    '''
    Declarations : Declaration Declarations
                 | empty
    '''
    if len(p) > 2:
        p[0] = Declarations(p[1], p[2])

def p_Declaration(p):
    '''
    Declaration : Type ID ';'
    '''
    p[0] = Declaration(p[2], p[1].upper())
```

```
def p_Statements(p):
    '''
    Statements : Statement Statements
               | empty
    '''
    if len(p) > 2:
        p[0] = Statements(p[1], p[2])

def p_Statement(p):
    '''
    Statement : Assignment
              | IfStatement
              | WhileStatement
              | ForStatement
              | ';'
              | Block
    '''
    p[0] = p[1]
```

Muchas de estas tienen un Expression que son operaciones que se deben resolver, bien pueden ser multiplicaciones, sumas, restas, divisiones, mods, negaciones (en el caso del signo "-" y de "!"), conjunciones ("&&"), relaciones ("==", "!=") y logical or ("||", la regla es expression en clite). Muchas de estas pueden hacerse con floats o con ints. Si se entrega un int y un float se convierten las dos a float.
Posteriormente se hizo un generador de código llvm con ayuda del patrón del patrón de diseño visitador y de la librería llvmlite. Este se encarga de "visitar" los nodos del árbol de sintaxis y crear código llvm que se puede ejecutar.
Para terminar se puede correr el código llvm con ayuda de la librería runtime.

Capacidades

Se pueden realizar asignaciones básicas:

```
data = '''

    int main() {
        int y;
        y = 5;
        return y;
    }
'''
```

```
define i32 @"main"()
{
entry:
  %"y" = alloca i32
  store i32 5, i32* %"y"
  %".3" = load i32, i32* %"y"
  ret i32 %".3"
}

5
```

Se pueden incluir operaciones matemáticas:

```
data = '''

    int main() {
        int y;
        y = 2 + 2 * 2;
        return y;
    }
'''
```

```
define i32 @"main"()
{
entry:
  %"y" = alloca i32
  %".2" = mul i32 2, 2
  %".3" = add i32 2, %".2"
  store i32 %".3", i32* %"y"
  %".5" = load i32, i32* %"y"
  ret i32 %".5"
}

6
```

Igualmente con floats (para que funcione todo lo relacionado con los floats se debe cambiar el CFUNCTYPE, véanse las limitaciones para más información):

```
data = '''

    float main() {
        float x;

        x = 2.0;
        x = x + 2.0;
        return x;
    }
'''
```

```
define float @"main"()
{
entry:
  %"x" = alloca float
  store float 0x4000000000000000, float* %"x"
  %".3" = load float, float* %"x"
  %".4" = fadd float %".3", 0x4000000000000000
  store float %".4", float* %"x"
  %".6" = load float, float* %"x"
  ret float %".6"
}

4.0
```

Se pueden definir ifs:

```
data =  '''

        int main() {
            int x;

            if (1 == 1 || 1 == 1 )
                x = 1;
            else
                x = 0;

            return x;
        }
        '''
```

```
define i32 @"main"()
{
entry:
  %"x" = alloca i32
  %".2" = icmp eq i32 1, 1
  %".3" = icmp eq i32 1, 1
  %".4" = or i1 %".2", %".3"
  br i1 %".4", label %"thenPart", label %"elsePart"
thenPart:
  store i32 1, i32* %"x"
  br label %"afterwards"
elsePart:
  store i32 0, i32* %"x"
  br label %"afterwards"
afterwards:
  %".10" = load i32, i32* %"x"
  ret i32 %".10"
}

1
```

Se pueden definir ciclos for:

```
data =  '''

        int main() {
            int x;
            int i;
            for(i = 0 ; i < 5; i = i + 1;){
                x = i;
            }

            return x;
        }
        '''
```

```
define i32 @"main"()
{
entry:
  %"x" = alloca i32
  %"i" = alloca i32
  store i32 0, i32* %"i"
  %".3" = load i32, i32* %"i"
  %".4" = icmp slt i32 %".3", 5
  br i1 %".4", label %"statementBody", label %"afterwards"
expressionBody:
  %".13" = load i32, i32* %"i"
  %".14" = icmp slt i32 %".13", 5
  br i1 %".14", label %"statementBody", label %"afterwards"
incrementBody:
  %".9" = load i32, i32* %"i"
  %".10" = add i32 %".9", 1
  store i32 %".10", i32* %"i"
  br label %"expressionBody"
statementBody:
  %".6" = load i32, i32* %"i"
  store i32 %".6", i32* %"x"
  br label %"incrementBody"
afterwards:
  %".16" = load i32, i32* %"x"
  ret i32 %".16"
}

4
```

Se pueden definir whiles:

```
data =  '''

        int main() {
            int x;
            int i;
            i = 1;
            x = 1;
            while(i <= 5){
                x = x * i;
                i = i + 1;
            }
            return x;
        }
        '''
```

```
define i32 @"main"()
{
entry:
  %"x" = alloca i32
  %"i" = alloca i32
  store i32 1, i32* %"i"
  store i32 1, i32* %"x"
  %".4" = load i32, i32* %"i"
  %".5" = icmp sle i32 %".4", 5
  br i1 %".5", label %"whileBody", label %"afterwards"
whileBody:
  %".7" = load i32, i32* %"x"
  %".8" = load i32, i32* %"i"
  %".9" = mul i32 %".7", %".8"
  store i32 %".9", i32* %"x"
  %".11" = load i32, i32* %"i"
  %".12" = add i32 %".11", 1
  store i32 %".12", i32* %"i"
  %".14" = load i32, i32* %"i"
  %".15" = icmp sle i32 %".14", 5
  br i1 %".15", label %"whileBody", label %"afterwards"
afterwards:
  %".17" = load i32, i32* %"x"
  ret i32 %".17"
}

120
```

Se pueden definir funciones:

```
data = '''
int return1() {
    int x;
    x = 1;
    return x;
}

int main() {
    int x;
    x = return1();
    return x;
}
'''
```

```
define i32 @"return1"()
{
entry:
  %"x" = alloca i32
  store i32 1, i32* %"x"
  %".3" = load i32, i32* %"x"
  ret i32 %".3"
}

define i32 @"main"()
{
entry:
  %"x" = alloca i32
  %".2" = call i32 @"return1"()
  store i32 %".2", i32* %"x"
  %".4" = load i32, i32* %"x"
  ret i32 %".4"
}

1
```

Se hace conversión de tipos:

```
data = '''
float main() {
    int x;
    float y;
    x = 1;
    y = 1.0;
    y = x + y;
    return y;
}
'''
```

```
define float @"main"()
{
entry:
  %"x" = alloca i32
  %"y" = alloca float
  store i32 1, i32* %"x"
  store float 0x3ff0000000000000, float* %"y"
  %".4" = load i32, i32* %"x"
  %".5" = load float, float* %"y"
  %".6" = sitofp i32 %".4" to float
  %".7" = fadd float %".6", %".5"
  store float %".7", float* %"y"
  %".9" = load float, float* %"y"
  ret float %".9"
}

2.0
```

Se pueden hacer operaciones con los resultados de las funciones:

```
data = '''
int fact(int t) {
    int x;
    int i;

    x = 1;
    i = 1;
    while(i <= 5){
        x = x * i;
        i = i + 1;
    }
    return x;
}


int main() {
    int x;
    x = fact(5) + fact(5);
    return x;
}

'''
```

```
define i32 @"fact"(i32 %".1")
{
entry:
  %"t" = alloca i32
  store i32 %".1", i32* %"t"
  %"x" = alloca i32
  %"i" = alloca i32
  store i32 1, i32* %"x"
  store i32 1, i32* %"i"
  %".6" = load i32, i32* %"i"
  %".7" = icmp sle i32 %".6", 5
  br i1 %".7", label %"whileBody", label %"afterwards"
whileBody:
  %".9" = load i32, i32* %"x"
  %".10" = load i32, i32* %"i"
  %".11" = mul i32 %".9", %".10"
  store i32 %".11", i32* %"x"
  %".13" = load i32, i32* %"i"
  %".14" = add i32 %".13", 1
  store i32 %".14", i32* %"i"
  %".16" = load i32, i32* %"i"
  %".17" = icmp sle i32 %".16", 5
  br i1 %".17", label %"whileBody", label %"afterwards"
afterwards:
  %".19" = load i32, i32* %"x"
  ret i32 %".19"
}
```

```
define i32 @"main"()
{
entry:
  %"x" = alloca i32
  %".2" = call i32 @"fact"(i32 5)
  %".3" = call i32 @"fact"(i32 5)
  %".4" = add i32 %".2", %".3"
  store i32 %".4", i32* %"x"
  %".6" = load i32, i32* %"x"
  ret i32 %".6"
}

240
```

Se pueden hacer negaciones:

```
data = '''
    int main() {
        int x;
        x = 1 + -1;
        return x;
    }

    '''
```

```
define i32 @"main"()
{
entry:
  %"x" = alloca i32
  %".2" = sub i32 0, 1
  %".3" = add i32 1, %".2"
  store i32 %".3", i32* %"x"
  %".5" = load i32, i32* %"x"
  ret i32 %".5"
}

0
```

Limitaciones

- No se puede hacer declaraciones con asignaciones (int x = 1). Esto debido a que se tendría que crear una regla que combinara las dos reglas.
- Por la limitación anterior no se puede definir el for de la forma for(int i = 0; i < x; i++)
- Los For se definen de la forma (x; x; x;) en vez de (x; x; x), esto debido a la forma en que está definida Assignment
- No se puede hacer x++ ni x += 1 ya que no existen esas reglas
- En las funciones se tienen que poner declarations, statements y return en ese orden. Esto debido a que no se tiene declarations, statements y return como parte de una clase más grande.
- Se tiene que colocar siempre el return al final por la misma.
- Se tiene que cambiar el CFUNCTYPE si se quiere ver un resultado for. No se encontró una forma directa de cambiarlo pero igualmente no tiene mucho caso ya que las funciones main en C están definidas para siempre regresar un int.
- No hay una forma de acceder a consola desde el código, esto porque se tendrían que hacer llamadas al kernel

Conclusiones individuales

Alejandro Castro Reus: La tarea de realizar un compilador fue una tarea compleja pero a la vez bastante gratificante. Esto debido a que tuvimos que aplicar conocimientos de otras clases como son las expresiones regulares, la administración de memoria y hasta los patrones de diseño.
Inicialmente se tuvieron que aplicar los conceptos vistos en clase como son el análisis léxico y los árboles de sintaxis para transcribir las reglas de clite a código y posteriormente para realizar reglas más complejas. Posteriormente nos enfrentamos a manejar toda la información de los nodos (por ejemplo en los argumentos de las funciones) con ayuda del patrón visitador. Por último se tuvo que hacer un análisis y testeo de las funciones muchas veces analizando el código llvm generado.

Es una actividad que hasta cierto punto es nostálgica tanto porque es de las últimas que realizamos como porque pudimos hacer un recorrido desde lo primero que vimos en la carrera que fue Python hasta nuestros conocimientos actuales y pasando por temas varios de la carrera. Igualmente es una actividad que me causó mucho interés ya que se ve lo interno de un lenguaje de programación y por eso me gustaría seguir aprendiendo más

José Antonio Bobadilla García: En mi opinión personal el desarrollar el compilador fue una tarea algo complicada ya que se me dificultaron los temas de expresiones regulares pero con el paso del tiempo y el seguir aprendiendo en las distintas clases me sentaron las bases como lo fue el uso de los visitantes para poder entender los temas necesarios para desarrollar y armar dicho compilador.

Me siento bastante bien conmigo mismo ya que como comenté, fue una experiencia algo difícil pero que al final de cuentas se logró y me llevo muchos aprendizajes con ella tanto para la vida profesional cómo personal.

Espero en el futuro poder aplicar los conocimientos adquiridos en el ámbito laboral el cuál está muy pronto por llegar y también estoy agradecido por los profesores por siempre brindarnos su apoyo para dichas tareas y lograr el objetivo tanto de la tarea como de la materia en sí.

Ituriel Mejia Garita: En el desarrollo de esta actividad, pude comprender cómo se complementan el análisis léxico y sintáctico para construir una herramienta de alto nivel como la que logramos desarrollar. Ha sido fascinante observar cómo los conceptos aprendidos en la primera parte del curso, centrados en el análisis léxico con expresiones regulares, evolucionaron para abarcar aspectos más amplios del contexto del código a través del análisis sintáctico.

Aunque el desarrollo de esta actividad ha sido desafiante, considero que el aprendizaje que obtuve es altamente enriquecedor. Valoro los conocimientos y habilidades adquiridos para llevar a cabo este tipo de análisis en lenguajes de programación. Además, ahora tengo una comprensión más profunda de la complejidad que hay detrás de un lenguaje de programación y cómo los compiladores lo interpretan.

Independientemente de la aplicación específica que realizamos en esta actividad, considero que estos conocimientos pueden extrapolarse a muchos otros proyectos y resultar útiles en diversas áreas de mi carrera. Por ejemplo, durante el aprendizaje de otros módulos y el desarrollo del reto, pude apreciar la importancia que el análisis léxico y el entendimiento de la gramática tuvieron al desarrollar una herramienta de detección de plagio junto con técnicas de machine learning. Este proceso de integración de conceptos me ha demostrado el valor de cómo estos temas se complementan para construir sistemas computacionales de alto nivel.