Compiladores
Proyecto 3: Generación de Código

Carlos Acosta Karla Esquivel
Yuan Yuan Luis Mayo
Andrea González

Facultad de Ciencias UNAM
2017-2

1. Preliminares

Para la realización de este proyecto, hicimos una reducción de nuestra gramática original planteada desde el proyecto 1, para generar código exclusivamente de expresiones aritméticas, álgebra booleana y asignaciones de variables. Asimismo, conservamos el operador de secuencialidad para no perder la oportunidad de manejar las variables al menos en el primer alcance.

1.1. La gramática reducida

Sea G la gramática de nuestro lenguaje de programación *KYC-IP*. Definiremos G, como la 4-tupla: $G = (N, T, P, S)$, con N, T, P conjuntos y $S \in N$ el símbolo de la producción inicial. A continuación, se muestran los conjuntos que la conforman:

```
N = {S, Prog, Prog', Inst, Expr, Bexp, Bterm, Beq, Bcom, Expr', Term, Factor,
Bool, Num, Decimal, Entero, Digito, Asig, Asig', Easig, Id, pos, Carac, Letra,
Tipo}
```

```
 T = {;, +, -, /, *, =, _, ==, >, <, >=, <=, and, or, not, (, ), int, float, bool,
0, 1,..., 9, a, ..., z, A, ..., Z} [1]
```

```
 P = {
     S → Prog
     Prog → Prog Prog'| Prog'
```

[1] En la gramática se utiliza "!" para representar a | con el propósito de no confundir el símbolo terminal "|" con la separación de las ramificaciones en las producciones.

```
Prog'→ Inst;
Inst → Expr | Asig
Expr → Bexp
Bexp → Bexp or Bterm | Bterm
Bterm → Bterm and Beq | Beq
Beq → Beq == Bcomp | Beq != Bcomp | Bcomp
Bcomp → Bcomp <Expr'| Bcomp >Expr'| Bcomp <= Expr'| Bcomp >= Expr'| Expr'
Expr'→ Expr'+ Term | Expr'- Term | Term
Term → Term * Factor | Term / Factor | Factor
Factor → Id | Num | (Expr) | - Factor | not Factor | Bool
Bool → true | false
Num → Entero Decimal
Decimal → . Entero | ε
Entero → Digito | Digito Entero
Digito → 0 | 1 | 2 | ... | 9
Asig → Easig | Asig'
Asig'→ Tipo Easig
Easig → Id = Expr | Id = Fun
Id → Letra Pos | Letra
Pos → Pos Carac | Carac
Carac → Letra | _ | Digito
Letra → a | b | ... | z | A | B | ... | Z
Tipo → int | float | bool
```

}

2. Compilación y ejecución del proyecto

El código fuente del proyecto, se encuentra dentro del directorio `src/`. Debe asegurarse que tenga esta forma mínima para su correcto funcionamiento:

```
src/
|-- ejemplo.kyc
|-- lexer.lex
|-- Makefile
|-- nodo.cpp
|-- nodo.h
|-- tabla.cpp
|-- tabla.h
|-- visitor.cpp
|-- visitor.h
|__ parser.y
```

2.1. Compilación del proyecto

En cualquier sistema operativo basado en Unix, con `g++`, `flex` y `bison` preinstalados, basta con ejecutar **make** en el directorio `src/` desde línea de comandos.

```
[user@host src]$ make
```

Dicha orden producirá un binario ejecutable de nombre ***kyc-ip*** listo para recibir código fuente de nuestro lenguaje.

En caso de existir un problema para generar dicho ejecutable, recomendamos volver a generar el código de los analizadores sintácticos y volver a intentar construir el ejecutable con las siguientes instrucciones:

```
[user@host src]$ make flex
[user@host src]$ make bison
[user@host src]$ make
```

2.2. Ejecución del analizador

Una vez obtenido el binario, basta:

```
[user@host src]$ ./kyc-ip <archivo>
```

Donde `<archivo>` es el archivo de texto claro con el código fuente correspondiente a nuestro lenguaje de programación.

Además hemos incluído una bandera en el ejecutable que permite generar archivos intermedios como son el *árbol de sintáxis abstracta* y la *tabla de símbolos*. Para obenterlos basta agregar a la orden de ejecución lo siguiente:

```
[user@host src]$ ./kyc-ip <archivo> -a
```

3. Resultado

Al término de la ejecución podremos encontrar nuevos archivos dentro de la carpeta de recursos del proyecto.

```
src/
|-- <archivo>.codigo
...
```

Si hemos elegido omitir la bandera de archivos intermedios. En otro caso, además del código ensamblador encontraremos los siguientes archivos.

```
src/
|-- <archivo>.codigo
|-- <archivo>.asa
|-- <archivo>.tds
...
```

3.1. Ejemplo

Por ejemplo para la siguiente **entrada**:

```
int w = 10 * 10 * 10;
float f = ( 56.0 / 89.098 );
bool d = true;
f = f + 10.0 * (10.0 + 11.0) / 76.7788 - 24.95;
d = d and d;
```

Obtendremos el siguiente **código objeto**:

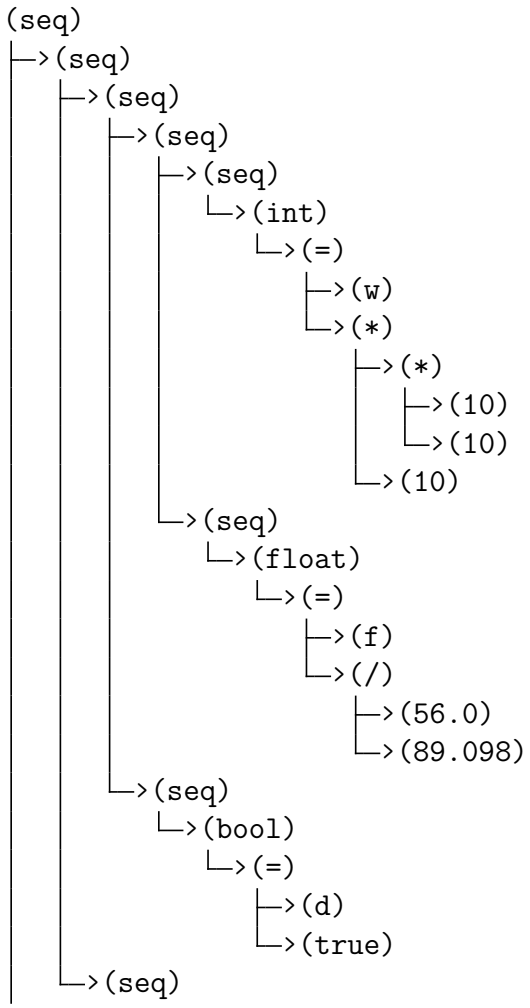
```
        .data
w:       .quad   0
f:       .float  0.0
d:       .quad   0
        .text
        .globl _start
_start:
movq 1000 %r0   #movq S, D D  S   LOAD
movq %r0 w      #movq S, D D  S   SAVE
movss 0.628521 %xmm0    #movss S, D D  S   LOAD
movss %xmm0 f   #movss S, D D  S   SAVE
movq 1 %r0      #movq S, D D  S   LOAD
movq %r0 d      #movq S, D D  S   SAVE
```

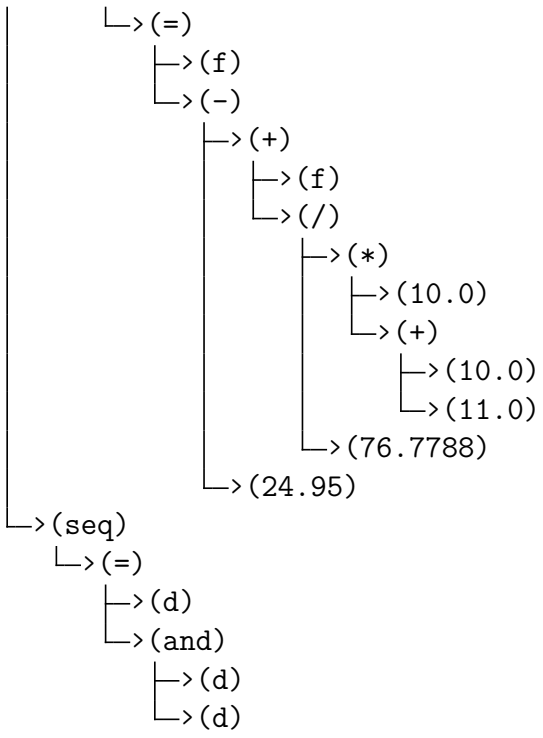
```
movss f %xmm1   #movss S, D D  S   LOAD
addss $2.735130 %xmm1   #addss S, D D  D + S
subss $24.95 %xmm1      #subss S, D D  D - S
movss %xmm1 f   #movss S, D D  S   SAVE
movq 0 %r2      #movq S, D D  S   LOAD
movq 0 %r1      #movq S, D D  S   LOAD
andq %r1 %r2    #andq S, D D  D and S
movq %r2 d      #movq S, D D  S   SAVE
```

La siguiente **tabla de símbolos**:

```
{bool: d, int: w, float: f}
```

Y el siguiente **ASA**:




└->(=)
├->(f)
└->(-)
├->(+)
├->(f)
└->(/)
├->(*)
├->(10.0)
└->(+)
├->(10.0)
└->(11.0)
└->(76.7788)
└->(24.95)
└->(seq)
└->(=)
├->(d)
└->(and)
├->(d)
└->(d)