Generador de código intermedio

Pablo Munoz Haro A01222422, Francisco Castellanos A01261268

October 24, 2018

Contents

1 Introducción

Este documento describe algunos aspectos de la arquitectura del programa `icg.py` que se encuentra anexo. Este programa es capaz de parsear un archivo fuente con sintáxis de lenguaje "c-minus" y produce el código intermedio correspondiente el cual es mostrado en stdout (aunque este puede ser redireccionado a algún archivo mediante la consola).

Para generar el código intermedio, el programa `icg.py` lee un archivo fuente y lo convierte en un token stream mediante un analizador léxico el cual utiliza un autómata finito en su implementación. Acto seguido, este stream de tokens es procesado por una serie de funciones recursivas, que en su conjunto conforman el generador de código intermedio. Cabe resaltar que para fines del programa `icg.py`, el análisis sintáctico no es llevado a cabo debido a que para fines de este proyecto se asume que los archivos fuente a ser procesados se encuentran correctos sintácticamente. El hecho de que no realizamos el análisis sintáctico tiene la consecuencia de que el código intermedio que generamos contiene errores en el orden de las llamadas

`begin_args` y `call` para los casos de llamadas a funciones que contienen parámetros que a su vez son llamadas a funciones, etc.

2 Análisis Léxico

Para producir el stream de tokens que consume nuestro generador de código intermedio se ejecuta un análisis léxico con el analizador léxico que desarrollamos para la primera parte del curso. Este analizador instancia una máquina de estados y va procesando cada caracter en el programa fuente. Cuando se encuentra un caracter unitario o un whitespace se genera el token que se ha estado acumulando durante la vida de la máquina de estados y posiblemente uno más (si es el caso de encontrar un unitario). Se implementa una lógica especial para el caracter `=`, ya que al encontrarse no se genera inmediatamente un token de asignación si no que primero se revisa si el de adelante también es un `=` en cuyo caso se genera el operador relacional de igualdad.

3 Generación de código intermedio

Nuestro generador de código intermedio está implementado en la clase `IntermediaCodeGenerator`, la cual consiste de los siguientes atributos:

- `tokens` Una lista de instancias de tokens (generado por el análisis sintáctico)
- `indent` Un número que representa el nivel de indentación del código (utilizado para imprimir el código intermedio de una manera más legible).
- `consumed_tokens` Una lista de los tokens que ya han sido consumidos por el generador del código intermedio. Conforme el código se va generando los tokens son sacados de la cabeza del atributo `tokens` y puestos en la cola de este atributo.
- `next_label` Un número que representa el componente numérico de la siguiente etiqueta que sea generada mediante el método `get_label()`
- `next_temp` Un número que representa el componente numérico de la siguiente variable temporal que sea generada mediante el método `get_temp()`
- `last_label` Una referencia a la última etiqueta generada por el método `get_label()`

- `last_temp` Una referencia a la última variable temporal generada por el método `get_temp()`.

Los métodos que conforman la clase `IntermediateCodeGenerator` son:

- `produce_triplet(a, b, c, d, e)` Produce una linea de código intermedio, en la implementación actual esto sólo conlleva imprimirlo a stdout, pero la clase podría ser extendida para que el código intermedio se almacenara en alguna estructura de datos.
- `consume_token()` Toma el primer token de la lista `tokens` y lo mueve al final de `consumed_tokens`. Esto es importante porque la familia de métodos `expand_<construct>` que se explicará a continuación revisa el primer token de `tokens` para determinar que tipo de código generar.
- `first_token()` Regresa el token en la cabeza de la lista `tokens`.
- `nth_token(n)` Regresa el enésimo token de la lista `tokens`.
- `put_back_token()` Des-consume el token consumido más recientemente, este método se implementó para permitir la posibilidad de hacer backracking, aunque no es utilizado en nuestra implementación.
- `get_label()` Regresa una string representando una etiqueta garantizada a ser única.
- `get_temp()` Regresa una string representando una variable temporal garantizada a ser única.
- `generate_code()` Método que inicia la generación de código intermedio.

Además de los métodos listados anteriormente, la clase contiene un gran número de métodos en de nombre `expand_<construct>`, donde `<construct>` es un símbolo de la gramática como `compound_statement` ó `selection_statement`.

El proceso de generación de código comienza al llamar el método `generate_code()` de una instancia de la clase `IntermediateCodeGenerator`. Este método lo único que hace es llamar el método `expand_function()` mientras la lista de `tokens` no se encuentre vacía. Esto lo hacemos debido a que sabemos de antemano que lo único que puede existir en el scope más externo de nuestros programas de c-minus son declaraciones de funciones. Los métodos de la familia `expand` hacen llamadas a `first_token()` o `nth_token()` para determinar que tipo de código debe generarse y luego hacen llamadas,

posiblemente intercaladas y recursivas a `produce_triplet()` para generar una línea de código intermedio o a otras funciones `expand` más especializadas. Por ejemplo, el método `expand_function()` consume el primer token (tipo de retorno de una función), el segundo token (nombre de la función), consume un token open parenthesis y después delega el procesado de sus parámetros a el método `expand_arguments()`. Después consume el token de paréntesis cerrado que delimita y genera el código intermedio `entry <funcname>`. Acto seguido consume la apertura de una llave y delega al método `expand_compound_statement()` la generación del código intermedio de el cuerpo de la función. Finalmente, la llave de cerradura que delimita la declaración de la función es consumida.

La gran mayoría de las funciones `expand` funcionan de la manera mencionada anteriormente, de una forma de "template". Con la finalidad de mantener este documento breve no explicamos cada función a detalle, pero invitamos al lector a dar un vistazo al código fuente. Las funciones sobre las que queremos profundizar son aquellas que se ejecutan de una manera distinta.

La función `expand` más distinta es `expand_simple_expression`. A pesar de que en la gramática que nos fue otorgada para realizar el proyecto el símbolo `simple_expression` puede generar más cosas, en nuestro código, cuando se llega a una expresión simple vamos a dejar de hacer más llamadas a funciones `expand` con la excepción de `expand_call` cuando una llamada a una función aparezca como operando de la expresión. Lo que `expand_simple_expression` va a hacer es poner la expresión simple en modo postfijo, y después generaremos el código intermedio para la expresión postfija. Cuando `expand_simple_expression` se encuentre con una llamada como operando, va a generar el código intermedio para la llamada, y luego reemplazar a la llamada con la variable temporal que esta regresa dentro de la lista postfija. Para ilustrar esto nos referimos al siguiente ejemplo:

Supongamos que nuestro programa llega a un punto donde va a expander la expresión simple `(8+3)*5 - fact(x)`. En este momento nuestro código va a crear una lista para almacenar la expresión postfija y otra para los operandos

```
postfix = []
operands = []
```

Después va a comenzar a consumir cada token, el cual será depositado en la lista correspondiente de acuerdo con las reglas de pasar una expresión infija a postfija

```
# parses (
postfix = []
operands = ['(']

# parses 8
postfix = ['8']
operands = ['(']

#parses +
postfix = ['8']
operands = ['(', '+']

# parses 3
postfix = ['8', '3']
operands = ['(', '+']

# parses )
postfix = ['8', '3', '+']
operands = []

# parses *
postfix = ['8', '3', '+']
operands = ['*']

# parses 5
postfix = ['8', '3', '+', '5']
operands = ['*']

# parses -
postfix = ['8', '3', '+', '5', '*']
operands = ['-']

# parses fact en este momento vemos que el operando es un
# identificador, vemos adelante y vemos que le sigue un paréntesis
# abierto (, entonces es una llamada, entonces generamos el código
# intermedio de la llamada y obtenemos su valor de retorno en la
# variable temporal t1, esto consumió los tokens de la llamada hasta
# el paréntesis de cierre que delimita los argumentos
postfix = ['8', '3', '+', '5', '*', 't1']
operands = ['-']
```

```
# no hay más tokens, se vacía los operands en la lista postfix
postfix = ['8', '3', '+', '5', '*', 't1', '-']
operands = []
```

Ahora que se ha generado la lisa postfija `['8', '3', '+', '5', '*', 't1', '-']`, mientras quede más de un token en la lista, se consumiran los primeros tres, se genera una variable temporal y una línea de código intermedio para ellos, y la variable temporal ocupará el lugar de los tres en la lista postfija como se muestra a continuación:

```
['8', '3', '+', '5', '*', 't1', '-']
# Se genera t2 = 8 + 3
['t2', '5', '*', 't1', '-']
# Se genera t3 = t2 * 5
['t3', 't1', '-']
# Se genera t4 = t3 - t1
[]
# fin
```

Las otras dos funciones `expand` "más creativas" son `expand_selection_statement` y `expand_iteration_statement`. Estas generan labels y código intermedio con gotos para poder alterar el flujo de ejecución del programa y hacer que el apuntador de código pueda brincar a una línea previamente ejecutada, o a una no inmediata, de manera que se ignoren cierto número de líneas intermedias.

4 Conclusión

La generación de código intermedio genera un código que es equivalente al programa fuente que se ha procesado. Este código es menos abstracto y por lo tanto es más verboso y utiliza menos primitivas. Para el caso de las llamadas a funciones, nuestro generador de código produce un código que puede ser considerado incorrecto, ya que por su naturaleza recursiva se generan instrucciones `begin_args` para las llamadas a función más externas, y luego para las más internas, pero lo que en verdad se quiere es que las llamadas internas sean procesadas primero y luego sus valores retornados utilizados como argumentos de las llamadas más externas. Para el caso de los `if` y los `while` la metodología recursiva de nuestro código funciona bien puesto que estos `statements` si operan evaluando primero los más exteriores.

5 Ejemplo

Consideremos el programa encontrado en el archivo `example.c`:

```
int fact( int x )
{ if (x > 1)
    return x * fact(x - 1);
  else
    return 1;
}

void main( void )
{ int x;
  x = read();
  if (x > 0) { write( fact(x) ); }
}
```

Después de correr

```
python icg.py example.c
```

Veremos en nuestra consola el siguiente output

```
entry fact
    t1 = x > 1
    if_false t1 goto L1
        begin_args
        t2 = x - 1
        param t2
        t3 = call fact 1
        t4 = x * t3
        return t4
    goto L2
    Label L1
        t5 = 1
        return t5
    Label L2
exit fact
entry main
    read x
    t6 = x > 0
```

```
    if_false t6 goto L3
        begin_args
        t7 = x
        param t7
        t8 = call fact 1
        t9 = t8
        write t9
    Label L3
    return
exit main
```