Universidad Nacional de Rio Cuarto

Facultad de Ciencias Exactas Físico-Químicas y Naturales

Departamento de computación

Taller de Diseño de Software (Cód. 3306)

Compilador C-TDS

Armas Lucas - Bentolila Fernando

2017

Análisis léxico y sintáctico

Tanto el léxico, como gramática del lenguaje fueron respetadas como se propusieron en un principio con el único cambio de permitir crear múltiples declaraciones de métodos y de variables, todo esto pactado con el profesor previamente, y se ha hecho uso de las herramientas lex/flex y yacc/bison para la construcción del programa en esta etapa, junto al lenguaje de programación C quien también será utilizado en etapas posteriores del proyecto.

El Analizador Léxico toma como entrada un archivo con extension “.ctds” que contiene el codigo fuente, y retorna tokens. Para la representación de los token se ha optado por un tipo “struct tokenLine” que contiene un campo que identifica a los símbolos del lenguaje, y otro que identifica la línea en la cual se encuentra el token, obteniendo una mayor facilidad a la hora de reportar errores.

Otra cosa a destacar, es que hemos hecho uso de un autómata para reconocer los comentarios del tipo /*...texto...*/ ya que estos presentaban un problema en el caso de los comentarios anidados.

```
%x comment
"/*"                BEGIN(comment);
<comment>[^*\n]*
<comment>"*"+[^*/\n]*
<comment>\n
<comment>"*"+"/"    BEGIN(INITIAL);
```

El Analizador Sintáctico, toma como entrada la secuencia de tokens y verifica que esta secuencia sea una secuencia valida dando como salida de esta etapa una tabla de símbolos (TDS), la cual es una lista de listas que se comporta como una pila para diferenciar niveles de visibilidad y alcance, las listas pueden contener variables y funciones del programa.
Los bloques de cada una de las funciones son representadas mediante un árbol sintáctico (AST) creado a partir de nodos, los cuales poseen los siguientes campos:

- tag: Permite identificar si es una variable, una constante, un operador o una función.
- noline: Contiene el número de línea pasado anteriormente por el *token* correspondiente.
- type: Campo que permite identificar el tipo del nodo siendo este un *int*, un *boolean*, o *void*.
- info: Es un puntero a una unión que contiene toda la información necesaria ya sea una variable, constante etc.
- También posee tres punteros a nodos del mismo tipo para poder formar el árbol sintáctico. En este punto se optó por tener tres hijos para abarcar correctamente el caso del *if-else*.

Otras consideraciones:

- Tanto en el análisis léxico como en el sintáctico, en el caso de identificar algún error, el programa se limitara a abortar inmediatamente su ejecución e informara sobre el correspondiente error.
- Se presentó un problema de precedencia y asociación con los operadores unarios NOT (!) y MENOS (-) el cual se solucionó con un operador ficticio UMINUS.

```
%right UMINUS
NOT expr %prec UMINUS
MENOS expr %prec UMINUS
```

Análisis Semántico

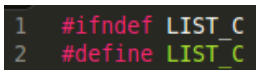
En esta etapa se verifica las reglas semánticas del lenguaje, por ejemplo, compatibilidad de tipos, visibilidad y alcance de los identificadores, etc.

En un comienzo se pensó en realizar el chequeo de tipos a medida que se creaban los AST en la etapa anterior, pero por cuestiones de diseño y de llevar a cabo cada etapa por separada, para lograr este objetivo hacemos uso de una función “checkType” el cual recibe como parámetro un AST de un método en particular. La forma de operar de esta función es realizando una búsqueda del tipo *deep-first-search;* cada vez que accede a un nodo, en el caso de ser este un tipo operador, se procede a comparar el tipo de este con el de sus hijos correspondientes, en caso de ser este una llamada a una función, se analiza si los parámetros pasados corresponden tanto en cantidad como en el tipo a los parámetros especificados por la función y en caso de esta retornar algún valor, compara el tipo del mismo con el tipo al que se lo asigna. Entonces, nuestra función de chequeo se encarga de asegurar que no existan errores de compatibilidad de tipos, con un adicional, también se encarga de registrar si se ha encontrado o no alguna palabra reservada “return [type]” haciendo uso de una variable global, ya que se dificultaba usarla localmente por culpa de la recursión. En caso de encontrar por lo menos un return por método, lo consideraremos aceptable.

El problema de visibilidad y alcance de los identificadores, es resuelto en la etapa anterior al mismo tiempo en el que se crea la tabla de símbolos, abriendo y cerrando nuevos niveles en la misma. Con una rápida búsqueda en esta tabla podemos determinar si se ha creado alguna función llamada “main”, caso contrario esto será reportado.

Otras consideraciones:

- Como en la etapa anterior, el programa suspenderá su ejecución en caso de encontrar algún error semántico retornando un informe sobre el mismo.
- Es algo de menor interés pero se produjeron varios problemas con la redefinición de tipos a la hora de compilar el programa, esto fue resuelto correctamente como lo veremos a continuación para el caso de List.c



Cuerpo de List.c

Generador Código Intermedio

En esta etapa del compilador, se obtiene como salida una representación intermedia (IR) del código. A partir de esta se generara el código objeto.

Se utilizara como código intermedio un Código de Tres Direcciones para las operaciones con tipos enteros y lógicos y para las operaciones de control de flujo.

Se ha optado por el uso de una lista simplemente enlazada para esta representación intermedia, conteniendo simplemente un código de tres direcciones y un campo que apunte al siguiente código. Para la representación de este código se utilizó un *struct* con cuatro campos siendo el primero el codigo de la operación a realizar y los restantes, pudiendo ser variables, constantes o temporales, representan los operando de la operación.

Todo este proceso es realizado por una función llamada “generateIC” que recorre los AST con una búsqueda *deep-first-search,* generando el código correspondiente a cada comando asociado por los nodos (operador) del árbol.

A continuación, se presenta la convención que se ha tomado para la representación del código.

Código de Tres Direcciones				
Instrucción				**Descripción**
Nombre	**Direc1**	**Direc2**	**Direc3**	
MOV	A	B	-	Mueve datos de (A ← B)
ADD	A	B	T	Suma enteros (T ← A + B)
SUB	A	B	T	Resta de enteros (T ← A – B)
MULT	A	B	T	Multiplicación de enteros (T ← A * B)
DIV	A	B	T	División de enteros (T ← A / B)
MOD	A	B	T	Módulo de enteros (T ← A % B)
MAY	A	B	T	Comparación por mayor (T ← A > B)
MIN	A	B	T	Comparación por menos (T ← A < B)
AND	A	B	T	Conjunción booleana (T ←A && B)
OR	A	B	T	Disyunción booleana (T ← A || B)
EQUAL	A	B	T	Comparación por igualdad (T ← A == B)
NEGB	A	-	T	Negación booleana (T ← !A)
NEGI	A	-	T	Negación de enteros (T ← -A)
RETURN	-	-	T	Salida de la funcion corriente con valor de retorno T
RETURNV	-	-	-	Salida de la funcion corriente
IFF	A	LABEL	-	Salta si la condición es falsa (if (!A) {salto a label})
LABEL	-	-	A	Crea una etiqueta
JMP	-	-	LABEL	Salto incondicional
LOAD	A	-	-	Prepara el parámetro para una próxima llamada a rutina
CALL	FUNC	-	T	Llamado a funcion
BEGIN	FUNC	-	-	Indica comienzo de la funcion
END	FUNC	-	-	Indica final de la funcion

A, B pueden ser variables, constantes o temporales.
T es un temporal auxiliar donde generalmente se guarda el resultado de una operación.
LABEL es una etiqueta (en la implementación es una variable).
FUNC es una función.

Generador Codigo Objeto

En esta etapa se genera código assembly x86-64 (sin optimizaciones) a partir de la lista de código intermedio resultante de la etapa anterior.

Se tomó la decisión de agregarle a las variables, temporales, y funciones un nuevo campo llamado “offset”, el cual facilitara el manejo de los mismos cuando necesitemos su localización en memoria, más concretamente en que posición del stack (offset(%ebp)) se encuentra y también para saber cuanto espacio reservar en memoria para el frame de cada método. El valor de este offset deberá ser un múltiplo de -8 y será asignado a medida que estas variables, temporales y funciones son creadas en nuestro árbol sintáctico, de igual forma en la etapa de generación de código intermedio cuando se crean nuevos temporales.

En el generador de código intermedio se modifico la instrucción “LOAD” para que acepte un segundo operando, el cual representará la posición que le corresponde al parámetro actual junto a los demás en la llamada a un método.

Con respecto al pasaje de parámetros, se tomo la decisión de que nuestros métodos acepten como máximo un total de seis parámetros, en donde deberán ser asignados de la siguiente manera:

- El parámetro 1 sera asignado al registro %rdi
- El parámetro 2 sera asignado al registro %rsi
- El parámetro 3 sera asignado al registro %rdx
- El parámetro 4 sera asignado al registro %rcx
- El parámetro 5 sera asignado al registro %r8
- El parámetro 6 sera asignado al registro %r9

Otra limitación del programa es que no permite pasar como parámetro de un método una variable definida globalmente.

Para finalizar, simplemente se realiza la traducción de cada instrucción de tres direcciones (generadas en la etapa anterior) a su equivalente en código assembly x86-64. Cabe destacar que este código objeto generado carece de eficiencia y elegancia, pero se realizaran optimizaciones al mismo en una próxima etapa.

Como comentario adicional, el codigo objeto generado variara levemente según el sistema operativo en el que nos encontremos, nuestro programa diferencia entre Mac y Linux gracias a un script.

IMPORTANTE!!

Actualmente existe un problema en la etapa de análisis sintáctico el cual provoca que seleccione reglas de gramática incorrectas cuando declaramos variables y funciones con tipos incorrectos por ejemplo: “voolean x;”, “pepe x”, ”nteger func()”, etc.
Se ha pactado (por decisión de los profesores) proseguir con el proyecto normalmente sin tener en cuenta este error, hasta poder identificar que es lo que lo provoca.

Optimizaciones

En esta etapa se propuso como una optimizacion tratar la propagacion de constantes, es decir, cuando veamos en nuestro árbol sintáctico una operación con todos sus operandos constantes, se procedera a realizar esa operación y retornar ese resultado como una nueva constante, en lugar de crear un nodo “operacion” con ambos operandos como se hacía previamente. Esta otimización no se realizara sobre el árbol sintáctico ya generado, sino que se creara el árbol directamente optimizado, realizando unos cambios en la etapa de análisis léxico y sintáctico donde se lo crea.

A continuación, veremos esto de forma más detallada con un ejemplo.

Dado el programa:

```
int main(){
    int x,z;
    z = 1;
    x = 1 + 1 + 1 + 1 + 1 + 1;
    return x;
}
```

Se obtuvieron los siguientes resultados:

Programa Sin Optimizar	Programa Optimizado

```
1     .globl _main
2  _main:
3     enter $64, $0
4     movq $1, -8(%rbp)
5     movq $1, %rax
6     addq $1, %rax
7     movq %rax ,-24(%rbp)
8     movq -24(%rbp), %rax
9     addq $1, %rax
10    movq %rax ,-32(%rbp)
11    movq -32(%rbp), %rax
12    addq $1, %rax
13    movq %rax ,-40(%rbp)
14    movq -40(%rbp), %rax
15    addq $1, %rax
16    movq %rax ,-48(%rbp)
17    movq -48(%rbp), %rax
18    addq $1, %rax
19    movq %rax ,-56(%rbp)
20    movq -56(%rbp), %rax
21    movq %rax, -16(%rbp)
22    movq -16(%rbp), %rax
23
24    leaq L_.str(%rip), %rdi
25    movq %rax, %rsi
26    movb $0, %al
27    callq _printf
28
29    leave
30    ret
```

```
1     .globl _main
2  _main:
3     enter $16, $0
4     movq $1, -8(%rbp)
5     movq $6, -16(%rbp)
6     movq -16(%rbp), %rax
7
8     leaq L_.str(%rip), %rdi
9     movq %rax, %rsi
10    movb $0, %al
11    callq _printf
12
13    leave
14    ret
```

Podemos apreciar una gran disminución, tanto de lineas de codigo como de espacio reservado en memoria para el frame, en el caso del programa optimizado.