Universidad Nacional de Río Cuarto
Facultad de Ciencias Exactas Físico-Químicas y Naturales
Departamento de Computación
Taller de Diseño de Software
(Cod. 3306)

Compilador de código C-TDS

Bongiovanni Ignacio
Año 2016

1. Descripción del lenguaje

El proyecto de la materia consiste en implementar un compilador para un lenguaje orientado a objetos simple, similar a C, llamado C-TDS.

Para las consideraciones del Léxico, Gramática y Semántica se siguieron los parámetros indicados en el documento de especificación del lenguaje provisto por la materia.

2. Etapas del Proyecto

2.1. Análisis Lexico-Sintactico.

Decisiones de Diseño:

- Los números reales no son discriminados por el Scanner, el mismo solo reconoce enteros, por lo que los reales son pasados al Parser como un entero (INT_LIT), seguido de un punto (DOT), y otro entero. Luego el Parser compone estos tres elementos en una regla que genera un no-terminal 'float_literal'.
- Las expresiones binarias (aritméticas, relacionales, comparaciones y condicionales) fueron agrupadas bajo la misma regla (expression) para simplificar las cuestiones de precedencia de operadores.

2.2. Análisis Semántico.

Estructura del Árbol Sintáctico Abstracto:

La estructura del AST para representar el cuerpo de un programa se realizó a partir del código y esquema provisto por la materia, realizando, cuando se consideró conveniente, el rediseño de las clases y jerarquías del mismo.
Junto a este documento se puede encontrar el diagrama de clases en formato UML que corresponde al paquete *ir.ast* (archivo 'ast-class-diagram.svg'). La clase raíz (AST) es de la cual extienden todos los demás componentes, y el segundo nivel de la jerarquía se compone de las siguientes clases:
Program, ClassDecl, Body, Declaration, Expression, Statement. Las primeras tres son clases concretas, mientras que las últimas son clases abstractas a partir de las cuales se definen el resto de los elementos.

Decisiones de Diseño:

- Las **declaraciones** se agruparon bajo una superclase llamada 'Declaration'. Las declaraciones que tienen un identificador se agruparon a su vez bajo una superclase denominada 'NamedDecl' que es descendiente directa de 'Declaration'.
- Los **tipos** se manejan mediante Strings, para permitir el uso de tipos definidos por el usuario. Se delega en la clase Type la capacidad de, mediante métodos estáticos (es decir que no se necesita instanciar un objeto de la clase Type), decidir cuando un tipo es básico, numérico, booleano, dos tipos son compatibles entre sí, etc.
- La **tabla de símbolos**, utilizada por el Builder Visitor, se comporta como una pila, donde cada elemento representa un nivel de ambiente de ejecución, donde se definen clases, atributos y métodos, parámetros, o variables locales según sea el tipo del nivel corriente.
- Los parámetros y las declaraciones internas a un método se encuentran a diferentes niveles, por lo que la declaración de una variable local del mismo nombre que un parámetro, hace inutilizable a este último.

En los visitantes:

Responsabilidades asignadas al Builder Visitor: Las cuestiones básicas de búsqueda y asignación de referencias entre locaciones y declaraciones de métodos, variables y arreglos. Además, por cuestiones de comodidad, se le asignaron las siguientes responsabilidades extras:
- Controlar que el número de argumentos en la llamada a un método sea el mismo que la cantidad de parámetros formales en la declaración.
- Al finalizar el proceso de búsqueda de referencias, se debe determinar si se definió una clase con nombre 'main'.

- Determinar que la clase main contenga un método 'main' y que el mismo sea de tipo void y no tenga parámetros.
- Controlar que los atributos de una clase sean de tipos básicos. Con el uso de la tabla de símbolos, se puede discriminar cuando el análisis de un elemento del tipo FieldDecl pertenece a la etapa de declaración de atributos o es la declaración de un field de variables locales. Por lo que, de ser el primer caso, se debe restringir el tipo de las declaraciones. Es por esto que esta tarea que podría pensarse como más apropiada para la etapa de chequeo de tipos se realiza en esta instancia.

Responsabilidades del TypeCheckVisitor:
Debe controlar el uso de tipos básicos para los retornos de los métodos y los parámetros de los mismos.
Que las asignaciones se hagan sobre tipos básicos (y en caso de que sean incrementos o decrementos se realicen sobre tipos numéricos).
Que el tipo de la expresión a asignar sea compatible con el tipo de la locación.
En las expresiones de retorno, se controla que respeten el contexto del método al que pertenecen (e.g. un return void es inválido en el contexto de un método de un tipo concreto, una expresión de tipo bool no puede ser retornada si el contexto no es de tipo bool).
Las condiciones de los if y los while deben ser de tipo bool.
Las sentencias break y continue no pueden estar fuera de un ciclo.
Las expresiones de un for, tanto inicialización como límite deben ser de tipo integer.
En las expresiones binarias y unarias: si es una expresión aritmética o relacional, sus operandos deben ser numéricos; si es una expresión lógica (AND, OR, NOT), sus operandos deben ser de tipo bool.
El tipo de cada argumento en la llamada a un método debe ser compatible con el del parámetro definido en la posición correspondiente.
Los métodos de tipo void no pueden ser utilizados como expresión.

Limitaciones encontradas:

No se logró determinar que un método de un tipo concreto termine siempre con una sentencia de return, es decir, no llegue al final de la declaración.

2.3. Generación de código intermedio.

Decisiones de diseño:

EL código intermedio a utilizar es el llamado codigo de tres direcciones o TAC, en el cual cada instrucción se compone de a lo sumo tres elementos (4 si incluimos el identificador de la instrucción), de los cuales dos juegan el rol de operandos y el tercero de resultado.
En la clase (enum) Inst se enumeran todas las posibles instrucciones a generar.
La clase TAC modela una instrucción de código de tres direcciones.
En el paquete semcheck se incluyo un visitante TACVisitor que se encarga de recorrer el AST y generar una lista de instrucciones del código intermedio.

2.4. Generación de código objeto.

Descripción del proceso:

La generación de código ensamblador se lleva a cabo una vez que se generó la lista de código de tres direcciones.
La clase AsmGen recibe dicha lista y traduce cada instrucción a código assembler x86_64 que es escrito sobre un archivo de texto plano con extensión .s, dicho archivo debe ser luego traducido a objeto y linkeado

al resto de las librerías de las que dependa, de esta última tarea se encarga enteramente el compilador de GNU (gcc).

Cuestiones encontradas y como se resolvieron:

- Chequeo de los indices al acceso sobre arreglos: se necesitó incrustar código assembler en cada acceso a arreglo donde se compara el resultado de la expresión con los limites del arreglo declarado.
- Declaración de objetos: los objetos son tratados como arreglos donde cada elemento es un atributo del mismo.
- Acceso a los atributos de un objeto: para acceder a un atributo de un objeto se necesita cargar la dirección efectiva del objeto declarado y mediante direccionamiento indirecto con el indice del atributo se accede al mismo.
- Atributos de la clase main: los atributos de la misma se tratan como estáticos y son declarados en el segmento .data del código assembler, la forma de acceder a los mismos difiere a la de los atributos de las demás clases.
- Desambiaguacion de métodos con el mismo nombre: En el chequeo de tipos (Type Check Visitor) antes de terminar el chequeo de una clase, se renombra cada uno de sus métodos (menos los externos y el método main) añadiendo el nombre de la clase seguida de un guion bajo al comienzo.

Limitaciones aún no resultas:

En los métodos con mas de 6 parámetros, del séptimo en adelante son ignorados.

2.4. Optimización.

2.4.1. Propagación de constantes

La optimizacion consiste en evaluar las operaciones binarias compuestas por números constantes directamente sobre el AST.

Un programa sin optimizar se ve de esta forma:

```
Class main {
  INT main() {
    INT  arr[10],;
    INT  x;
    arr[3 * 2] = 4 - 2 + 5 * 3;
    x = arr[3 + 3];
    return 0 * 0;
  }
}
```

Mientras que optimizado se ve así:

```
Class main {
  INT main() {
    INT  arr[10],;
    INT  x,;
    arr[6] = 17;
    x = arr[6];
    return 0;
  }
}
```

2.4.2. Reutilización de offset

Los lugares reservados para las variables internas a un bloque son reasignados luego para variables de bloques externos al mismo.

Programa sin optimizar

```
integer main(){
    integer x; //Offset=8
    if (true){  //Offset=16
        integer y; //Offset=24
        integer z; //Offset=32
        if (false) //Offset=40
        {
            integer a; //Off= 48
            integer b; //Off= 56
        }
    }
    else{
        integer w; //Offset=64
    }
}
```

```
main:
    enter $64,$0
    # [ DEC-VAR-INT     | x | - | - ]
    movq $0, -8(%rbp)
    # [ LC-BOOL      | true | - | t1 ]
    movq $1, -16(%rbp)
    # [ JF       | ElseIf1 | t1 | - ]
    mov -16(%rbp), %r11
    cmp $0, %r11
    je ElseIf1
    # [ DEC-VAR-INT     | y | - | - ]
    movq $0, -24(%rbp)
    # [ DEC-VAR-INT     | z | - | - ]
    movq $0, -32(%rbp)
    # [ LC-BOOL      | false | - | t2 ]
    movq $0, -40(%rbp)
    # [ JF       | ElseIf2 | t2 | - ]
    mov -40(%rbp), %r11
    cmp $0, %r11
    je ElseIf2
    # [ DEC-VAR-INT     | a | - | - ]
    movq $0, -48(%rbp)
    # [ DEC-VAR-INT     | b | - | - ]
    movq $0, -56(%rbp)
    # [ JMP      | EndIf2 | - | - ]
    jmp EndIf2
    # [ LABEL    | ElseIf2 | - | - ]
    ElseIf2:
    # [ LABEL    | EndIf2 | - | - ]
    EndIf2:
    # [ JMP      | EndIf1 | - | - ]
    jmp EndIf1
    # [ LABEL    | ElseIf1 | - | - ]
    ElseIf1:
    # [ DEC-VAR-INT     | w | - | - ]
    movq $0, -64(%rbp)
    # [ LABEL    | EndIf1 | - | - ]
    EndIf1:
    # [ METHOD-END   | - | - | - ]
    leave
    ret
```

Programa optimizado

```
integer main(){
    integer x; //Offset=8
    if (true){  //Offset=16
        integer y; //Offset=24
        integer z; //Offset=32
        if (false) //Offset=40
        {
            integer a; //Off= 48
            integer b; //Off= 56
        }
    }
    else{
        integer w; //Offset=24
    }
}
```

```
main:
    enter $56,$0
    # [ DEC-VAR-INT     | x | - | - ]
    movq $0, -8(%rbp)
    # [ LC-BOOL      | true | - | t1 ]
    movq $1, -16(%rbp)
    # [ JF       | ElseIf1 | t1 | - ]
    mov -16(%rbp), %r11
    cmp $0, %r11
    je ElseIf1
    # [ DEC-VAR-INT     | y | - | - ]
    movq $0, -24(%rbp)
    # [ DEC-VAR-INT     | z | - | - ]
    movq $0, -32(%rbp)
    # [ LC-BOOL      | false | - | t2 ]
    movq $0, -40(%rbp)
    # [ JF       | ElseIf2 | t2 | - ]
    mov -40(%rbp), %r11
    cmp $0, %r11
    je ElseIf2
    # [ DEC-VAR-INT     | a | - | - ]
    movq $0, -48(%rbp)
    # [ DEC-VAR-INT     | b | - | - ]
    movq $0, -56(%rbp)
    # [ JMP      | EndIf2 | - | - ]
    jmp EndIf2
    # [ LABEL    | ElseIf2 | - | - ]
    ElseIf2:
    # [ LABEL    | EndIf2 | - | - ]
    EndIf2:
    # [ JMP      | EndIf1 | - | - ]
    jmp EndIf1
    # [ LABEL    | ElseIf1 | - | - ]
    ElseIf1:
    # [ DEC-VAR-INT     | w | - | - ]
    movq $0, -24(%rbp)
    # [ LABEL    | EndIf1 | - | - ]
    EndIf1:
    # [ METHOD-END   | - | - | - ]
    leave
    ret
```