Informe Compiladores

Taller de Diseño de Software [3306]

Marcial, Valentin
Rosatti, Nicolle
Baldevenito, Joaquin

1. Introduccion

Este proyecto trata de la creacion de un compilador gcc. El informe se basará en como fue el paso a paso de la creacion del proyecto, su forma de uso, y el contenido aplicado de la materia. Cada seccion es una de las entregas realizadas.
El objetivo del informe es realizar una breve descripcion de lo llevado a cabo, contar un poco de las formas en las que fuimos resolviendo los problemas y la eleccion de las distintas estructuras implementadas. Tambien hablaremos sobre los distintos desafios que se nos cruzaron en el trayecto de la realizacion del mismo. Dejando al final una breve conclusion de todo lo aprendido.

2. Análizador Sintáctico y Léxico (Scanner y Parser)

2.1. Creacion de archivo flex y bison

Para comenzar con el proyecto utilizamos las herramientas flex que es un lexer y bison que es un parser LLR.

Dentro de nuestro archivo flex.l creamos los tokens de nuestro lenguaje y dentro de nuestro bison.y creamos las gramaticas del mismo.

En este paso dentro de nuestro archivo flex.l creamos lo tokens que son utilizados en el archivo bison.y. Aca decidimos colocar las operaciones principales y los tipos basicos como tokens.

Para los terminales TRUE y FALSE decidimos trabajarlos como enteros 1 y 0 respectivamente.

En nuestro caso el manejo de comentarios nos salio a la primera y los representamos de esta forma:

1. `"//".*`
2. `"/*"([^*]|\*+[^*/])*\*+"/"`

Luego de plantear las gramaticas comenzamos a crear el arbol sintactico, para esto creamos nuestros Symbol que son un tipo enumerado que contiene

1. SymbolType type: El tipo del simbolo
2. SymbolKind kind: Si el simbolo es VAR o FUNC
3. char *name: El nombre del simbolo
4. Valores valor: Que valor es (numero o id)
5. struct Tree *node: El nodo que representa dentro del arbol sintactico

3. Analizador Semantico

3.1. Creacion de Tabla de simbolos y alcance

En esta etapa realizamos la creacion de una tabla que contiene los simbolos. En esta tabla tambien se realizo el manejo del alcance y la creacion de ambientes.

La SymbolTable es un registro de la forma

1. Symbol **symbols: Es la referencia a la referencia de los simbolos
2. int size: Es el tamaño de la tabla
3. int capacity: La capacidad total de la tabla

Luego comenzamos a realizar el checkeo de tipos acompañado del alcance, implementamos un alcance estatico. Este mismo fue implementaado con una pila de tablas de simbolos.

Adicionalmente, se implementó una nueva validación semántica: ahora se prohíbe la declaración de variables globales después de la definición de la función `main`. El analizador lleva un registro de si `main` ya ha sido declarada y, si encuentra una declaración global posterior, emite un error semántico.

Para el checkeo de tipos recorrimos el arbol sintactico y fuimos checkeando en cada declaracion y asignacion que los simbolos existan. En este paso tuvimos complicacion con la forma en la que estaban definidas las declaraciones de los metodos ya que pueden contener parametros. El problema que tuvimos era que al realizar la comparacion entre la cantidad de parametros y los tipos no referenciabamos bien al arbol entonces se nos perdia la referencia, para solucionar esto nos ayudamos de bocetos del arbol para entender mejor el acceso que necesitabamos y la forma de recorrer el arbol para ir viendo el paso a paso de como se comparaban entre el subarbol de definicion del metodo con el subarbol de la llamada al metodo.

Para las pruebas creamos distintos archivos que se clasifican por nombre para corroborar si fallan cuando tienen que fallar y si estan correctos cuando se debe. Todo este manejo lo realizamos a travez de un archivo principal main y un makefile donde especificamos las distintas formas de correr el programa. Los resultados de estos test son guardados en una carpeta principal resultados en donde estan separados por carpetas por tipos de test.

Flags para ejecucion:

1. -o `<salida>` Renombra el archivo ejecutable a `<salida>`(archivo de salida).
2. -t `<etapa>` (`<etapa>` es una de scan, parse, codinter o assembly), la compilación procede hasta la etapa dada.
3. -opt (Realiza optimizaciones, all ejecuta todas las optimizaciones soportadas).
4. -d (Imprime información de debugging. Si la opción no es dada, cuando la compilación es exitosa no debería imprimirse ninguna salida).

Formas de ejecutar archivos especificos:

1. ./c-tds archivo.ctds
2. ./c-tds –target `<etapa>` archivo.ctds

Ejemplo:

```
./c-tds -d --target scan tests/correct/TestCorrect1.ctds
```

Formas de ejecutar con makefile:

1. make `run_tests TEST_TARGET=parse` (Por defecto se corre con scan, y de esta forma se ejecutan todos los test creados)

Ejemplo de la estructura del arbol:

```
    Árbol antes de ejecutar asignaciones:
    CODE
      left:
        METHOD(Symbol: print_int, type=2, value=0)
          left:
            METHOD_HEADER
              left:
                ID(Symbol: print_int, type=2, value=0)
                  left:
                    T_VOID
              right:
                ARGS
                  left:
                    LIST
                      left:
                        DECLARATION(Symbol: i, type=0, value=0)
                          left:
                            T_INT
      right:
        CODE
          left:
            METHOD(Symbol: get_int, type=0, value=0)
              left:
                METHOD_HEADER
                  left:
                    ID(Symbol: get_int, type=0, value=0)
                      left:
                        T_INT
                  right:
                    ARGS
            right:
              CODE
                left:
                  METHOD(Symbol: main, type=2, value=0)
                    left:
                      METHOD_HEADER
                        left:
                          ID(Symbol: main, type=2, value=0)
                            left:
                              T_VOID
                        right:
                          ARGS
                    right:
                      BLOCK
                        left:
                          LIST
                            left:
                              DECLARATION(Symbol: y, type=0, value=0)
                                left:
                                  T_INT
                                right:
                                  METHOD_CALL(Symbol: get_int, type=3, value=0)
...
```

Ejemplo de Error semantico:

```
    Árbol antes de ejecutar asignaciones:
    CODE
...
            METHOD(Symbol: main, type=0, value=0)
...
                BLOCK
                  left:
                    LIST
                      left:
                        DECLARATION(Symbol: x, type=0, value=0)
...
                      right:
                        LIST
                          left:
                            DECLARATION(Symbol: x, type=0, value=0)
...
                              right:
                                -> ERROR en la línea 5: Redeclaración de 'x'
    Error semántico
                                INT(Symbol: anon, type=0, value=20)
...
```

Ejemplo de Error sintactico:

```
    -> ERROR sintáctico en la línea 2: syntax error. Token actual: 'y'
    Error en el parseo
```

4. Generador Código Intermedio

4.1. Crear las estructuras

En este paso creamos nuestro archivo GenCode que consta de tres estructuras

1. IRInstr: contiene todos los identificadores para saber que instruccion se debe ejecutar
2. IRCode: es la estructura principal que contiene las cosas del codigo de tres direcciones
   a) IRInstr op: contiene la operacion que se va a realizar
   b) Symbol *arg1: Es el primer argumento para la operacion
   c) Symbol *arg2: Es el segundo argumento para la operacion
   d) Symbol *result: Es el resultado de op con arg1 y arg2
3. IRList: Es la lista que contiene las instrucciones a realizar

4.2. Enumerado para las instrucciones

En esta seccion comenzamos a agregar las operaciones que utilizaremos para generar el codigo intermedio

```
static const char *ir_names[] = {
    "LOAD","DECL", "STORE","STORAGE", "ADD","SUB", "UMINUS", "MUL","DIV","MOD",
    "AND","OR","NOT",
    "EQ","NEQ","LT","LE","GT","GE",
    "LABEL","GOTO", "RET", "PARAM", "CALL", "METHOD", "F_METHOD", "METH_EXT", "PRINT"
};
```

4.3. Generar instrucciones por casos

Teniendo la constante del paso anterior lista con todas las operaciones assembler comenzamos a desarrollar para cada operacion base el conjunto de instrucciones en assembler correspondiente. La función principal es `gen_code`, que recorre el Árbol Sintáctico Abstracto (AST) de forma recursiva. Utiliza un `switch` para manejar cada tipo de nodo (`NODE_TIPO`).

La función devuelve un `Symbol*` que representa el "lugar"(un registro temporal) donde se almacenó el resultado de una expresión. Para las sentencias (como `NODE_IF` o `NODE_DECLARATION`), devuelve `NULL`.

Manejo de Literales y Temporales: Para manejar literales (como números) y resultados intermedios, usamos símbolos temporales. La función `newTempSymbol()` crea un nuevo temporal (ej. `t0`, `t1`, ...). Para un literal (ej. `NODE_INT`), usamos la instrucción `IR_STORAGE` para mover el valor literal a un nuevo temporal en la pila. El `switch` contiene un caso para `NODE_INT`, `NODE_TRUE` y `NODE_FALSE`, que crean un temporal, emiten `IR_STORAGE` para guardar el valor en él y devuelven dicho temporal.

Operaciones Binarias: Para operaciones como la suma (`NODE_SUM`), la función genera recursivamente el código para los hijos izquierdo y derecho (`l` y `r`). Luego, crea un nuevo temporal `t` para el resultado y emite la instrucción `IR_ADD(l, r, t)`. Este patrón se repite para `SUB`, `MUL`, `DIV`, `AND`, `OR`, etc.

Manejo de Llamadas a Métodos: Este es un caso complejo. Para un `NODE_METHOD_CALL`, primero se utiliza una función helper `gen_method_args`. Esta función recorre los argumentos de derecha a izquierda (para la convención de C), pero emite las instrucciones `IR_PARAM` con el índice correcto (de izquierda a derecha). Una vez procesados todos los argumentos, `gen_code` emite la instrucción `IR_CALL` con el nombre de la función y un nuevo temporal para guardar el valor de retorno.

Control de Flujo (If-Else): Para el control de flujo, generamos nuevas etiquetas usando `newLabel()`. En un `NODE_IF_ELSE`, se genera el código de la condición, seguido de un `IR_GOTO` condicional (que salta si la condición es falsa) a una etiqueta `label_else`. Luego, se genera el código del bloque `then`, seguido de un `IR_GOTO` incondicional a `label_end`. Se emite `label_else`, se genera el bloque `else`, y finalmente se emite `label_end`. El caso `NODE_WHILE` es similar, usando etiquetas para crear un bucle.

Declaraciones (Globales y Locales): El manejo de `NODE_DECLARATION` se ha vuelto más sofisticado. El generador ahora diferencia entre inicialización estática (con un valor constante, ej. `int x = 5;`) e inicialización dinámica (con una expresión, ej. `int x = a + b;`).

- **Inicialización Estática Global:** Se emite una única instrucción `IR_DECL` que lleva tanto la variable como su valor constante (en `arg1`). El backend se encargará de esto.
- **Inicialización Dinámica Global:** Se emiten dos instrucciones: primero un `IR_DECL` (con valor `NULL` en `arg1`) para reservar el espacio en la sección `.bss`, y luego, en el flujo de ejecución, un `IR_STORE` para asignar el resultado de la expresión a esa variable.
- **Variables Locales:** Se tratan como inicialización dinámica, generando un `IR_STORE` en la pila.

4.4. Asignación de Offsets para Temporales

Después de generar todo el código IR, ejecutamos una pasada final `offset_temps` sobre la `IRList`. Esta función es crucial para la generación de assembly. Recorre la lista de instrucciones IR, identifica el inicio de cada método (`IR_METHOD`) y, a partir del espacio ya reservado para parámetros y locales (calculado en la etapa de Assembly), asigna offsets en la pila (relativos al `%rbp`) para todos los símbolos temporales (`t0`, `t1`, ...) que se crearon durante la generación de código intermedio. Al final del método (`IR_FMETHOD`), actualiza el espacio total del stack del método para incluir estos temporales.

5. Generador de Assembler

La etapa final del compilador es el Generador de Assembler. Este componente toma la lista de Código Intermedio (`IRList`) y la traduce a código Assembly x86-64.

5.1. Cálculo de Offsets y Layout del Stack

Antes de la generación de assembly, se realiza un paso crucial: `calculate_offsets`. Esta función recorre el AST (no el IR) para determinar el layout del stack para cada método. Asigna un offset (desplazamiento) relativo al puntero base (`%rbp`) a cada parámetro y variable local.

La organización del stack frame que implementamos es la siguiente:

```
* LAYOUT DEL STACK (para métodos):
* 16(%rbp)     <- parámetro 7 (si existe)
* 8(%rbp)      <- return address
* 0(%rbp)      <- previous %rbp
* -8(%rbp)     <- parámetro 1 (guardado desde %rdi)
* -16(%rbp)    <- parámetro 2 (guardado desde %rsi)
* ...
* -X(%rbp)     <- variables locales del método
* -Y(%rbp)     <- variables de bloques anidados (if, while, etc.)
* -Z(%rbp)     <- temporales del código intermedio
```

La función `calculate_offsets_helper` maneja esta lógica, recorriendo el AST. Al encontrar un `NODE_METHOD`, itera sobre sus parámetros. Los primeros 6 reciben offsets negativos (ej. `-8(%rbp)`) ya que se guardarán desde registros. A partir del séptimo, reciben offsets positivos (ej. `16(%rbp)`) ya que ya están en la pila. Luego, continúa asignando offsets negativos crecientes a las variables locales y de bloques anidados.

5.2. Manejo de Variables Globales (.data vs .bss)

Una mejora crucial en el generador de assembly es la correcta separación de variables globales en las secciones `.data` y `.bss`. La función `generateAssembly` primero llama a `collect_globals` para recolectar todas las declaraciones `IR_DECL` y `IR_STORE` globales, evitando duplicados.

Luego, llama a `print_global_sections`. Esta nueva función recorre la lista de variables globales recolectadas **dos veces**:

1. **Pasada .data:** En la primera pasada, imprime la cabecera `.data` y luego solo imprime las variables que tienen un valor de inicialización explícito y **distinto de cero** (ej. `var: .quad 5`). Esto lo sabe gracias a que `IR_DECL` guardó el valor constante en `arg1`.

2. **Pasada .bss:** En la segunda pasada, imprime la cabecera `.bss` y luego imprime todas las variables que no fueron inicializadas (valor `NULL`) o que fueron inicializadas a cero. Para esto, utiliza la directiva `.comm` (ej. `.comm var, 8`) para reservar espacio en la sección de "Block Started by Symbol".

5.3. Generación de Instrucciones (Traducción de IR)

La generación de assembly se inicia con `generateAssembly`, que (después de manejar las globales) imprime la cabecera `.text` e itera sobre la `IRList`, llamando a `generateInstruction` para cada instrucción de IR. Se mantiene un puntero al `current_method` para dar tratamiento especial a `main`.

La función `generateInstruction` es un gran `switch` que mapea cada `IRInstr` a una función helper específica que imprime el código assembly (ej. `IR\_ADD` llama a `generateBinaryOp(inst, "addq")`).

5.4. Ejemplos de Traducción de IR a Assembly

Prólogo y Epílogo de Métodos (con manejo de 'main'): Al encontrar `IR_METHOD`, se genera la etiqueta y se llama a `generateEnter`. Esta instrucción imprime `enter` $(\%d), \$0$, donde `%d` es el espacio total del stack calculado (locales + temporales), asegurando el alineamiento a 16 bytes.

Al encontrar `IR_RETURN`, `generateReturn` ha sido mejorada. Ahora recibe el contexto del método actual. Si el método es `main`:

- Si hay un valor de retorno explícito (ej. `return 5;`), se mueve a `%rax` como de costumbre.
- Si **no** hay valor de retorno (un `return;` o el final de la función), la función **fuerza un valor de 0** (`movq $0,%rax`) en el registro de retorno. Esto asegura que el programa principal siempre devuelva un código de salida `0` (éxito) por defecto, cumpliendo con la convención estándar del sistema operativo.

Finalmente, se imprimen las instrucciones `leave` y `ret` para deshacer el stack frame y volver.

Operaciones Binarias y Manejo de Errores (Div/Mod): `generateBinaryOp` maneja `addq`, `subq`, `imulq` de forma estándar, usando `%rax` como registro acumulador. Para `idivq` (división) y `modq` (módulo), implementamos un chequeo de seguridad. Antes de la división, se carga el divisor en `%rcx` y se compara con cero (`cmpq $0, %rcx`). Si es cero, se salta (`je`) a una etiqueta de error (`_division_by_zero_error_N`) que llama a `exit(136)`. Si no es cero, se realiza la operación `idiv` y se guarda el resultado (`%rax` para div, `%rdx` para mod).

Manejo de la Convención de Llamadas (Calling Convention): El manejo de llamadas es la parte más compleja y se divide en tres funciones que implementan la convención x86-64:

1. `generateParam` (Lado del llamador): Lee una instrucción `IR\_PARAM`. Según el índice del parámetro, imprime `movq` del temporal en la pila al registro correcto (ej. `movq -X(%rbp), %rdi` para el parámetro 0) o `pushq` si es el 7mo argumento o más.
2. `generateSaveParam` (Lado del llamado): Al inicio de la función (`IR\_METHOD`), se generan `IR\_SAVE\_PARAM` para los primeros 6 argumentos. Esta función traduce eso a `movq` desde el registro a su "home slot.[en] la pila (ej. `movq %rdi, -8(%rbp)`).
3. `generateCall` (Lado del llamador): Emite la instrucción `call` y, si la función tiene un valor de retorno, guarda el resultado de `%rax` en el temporal de destino correspondiente.