Metodología empleada y manual de operación

Diseño de compiladores
Dr. Edgar E. Vallejo

Saúl de Nova Caballero (A01165957)
Diego Galíndez Barreda (A01370815)

5 de mayo de 2016

1. Metodología empleada

A continuación viene un resumen de los cambios que se hicieron a la especificación para poder cumplir todos los requerimientos.

1.1. Expresiones regulares

Para las expresiones regulares, se convirtieron todos los operadores, tipos y palabras claves del lenguaje en tokens. También se verificaron constantes de strings, enteros en hexadecimal y enteros, flotantes en notación científica y sin la notación. Se consideraron todas las cadenas que no tengan saltos de línea como válidas. Se verificó para los identificadores que fueran a lo más de 31 caracteres. En el caso de que un caracter no sea ' ', \n, \r ó \t.

Para acomodar comentarios, se utilizó la instrucción **BEGIN** para determinar los estados de flex. Se creó el estado **C_COMMENT** para determinar cuando te encuentras dentro de un comentario.

1.2. Modificaciones a la gramática

Para manejar los bloques de código en las instrucciones adecuadas, if, for, $while$, $block$, se insertaron expresiones *scopeStart* y *scopeEnd* para manejar la profundidad de las tablas de símbolos.

Para poder tener las expresiones con $^+$ y * se descompusieron las expresiones de la siguiente manera:

$Expr ::= Term^+$
$Expr ::= Term \,|\, Expr\, Term$

se cambió a:

$Expr ::= Term^*$
$Expr ::= \epsilon \,|\, Expr\, Term$

Con el propósito de generar el código intermedio correcto indexando, arreglos y *strings*, se cambió valorL y expr. El valor original era:

$Expr ::= ValorL \mid ValorL = Expr$
$ValorL ::= identificador \mid Expr\,[\,Expr\,]$

se cambió a:

$Expr ::= ValorL \mid ValorL = Expr \mid ValorL\,[\,Expr\,] \mid ValorL\,[\,Expr\,] = Expr$
$ValorL ::= identificador$

Para poder hacer los reales y los formales, se separó la gramática en 3 para poder acomodar comas, vacíos y final sin comas.

$Reals ::= Expr^{+}\,, \mid \epsilon$

se cambió a:

$Reals ::= \epsilon \mid RealsDefEnd$
$RealsDefEnd ::= expr \mid RealsDef\, expr$
$RealsDef ::= expr\,, \mid RealsDef\, expr\,,$

Para finalizar, en las funciones se nos presentó un problema. Para poder asegurarnos que los tipos de los *returns* fueran correctos tuvimos que cambiar la gramática de esto:

$DeclFuncion ::= Tipo\, identificador\,(Formales)\, BloqueInstr$
$\mid void\, identificador\,(Formales)\, BloqueInstr$

a:

$DeclFuncion ::= TipoFuncion\,(Formales)\, BloqueInstr$
$TipoFuncion ::= tipo\, identificador \mid void\, identificador$

Este cambio puede parecer minúsculo, sin embargo, nos permite que la tabla de símbolos guarde el identificador de la función para ser usado en la función de abajo.

Para manejar los *fors* se utilizó la técnica de *backpatching* y se cambió la gramática insertando expresiones vacías, cuyo único propósito es generar código. Por lo que las expresiones:

$InstrIf ::= if\,(\,Expr\,)\, Instr < else\, Instr >$
$InstrWhile ::= while\,(\,Expr\,)\, Instr$
$InstrFor ::= for\,(< expr >;\, Expr\,;< expr >)\, Instr$

se cambiaron a:

$InstrIf ::= if\,(\,\textbf{InstrIfCond}\,)\, Instr\, \textbf{InstrIfEnd}$
$InstrIf ::= if\,(\,\textbf{InstrIfCond}\,)\, Instr\, \textbf{InstrIfElse}\, else\, Instr\, \textbf{InstrIfElseEnd}$
$InstrWhile ::= \textbf{InstrWhileStart}\, while\,(\,\textbf{InstrWhileCond}\,)\, Instr\, \textbf{InstrWhileEnd}$
$InstrFor ::= for\,(< expr > \textbf{InstrForStart}\,;\, \textbf{InstrForCond}\,;< expr >)$
$> \quad \textbf{InstrForLabel}\, Instr\, \textbf{InstrForEnd}$

1.3. Tabla de símbolos

Para poder manejar adecuadamente los scopes del programa fue necesario utilizar una tabla recursiva de símbolos. Se muestra el concepto en la figura 1.

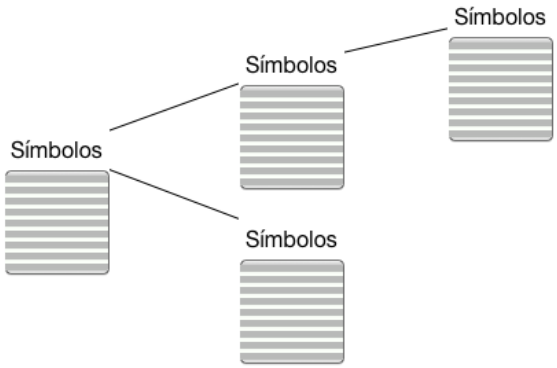

Figura 1: Tabla recursiva de símbolos

En el momento en que se cambia el scope (**instrIf**, **instrWhile**, **instrFor**, **declFunction**), se crea una nueva tabla de símbolos. Todas las tablas tienen una tabla padre, por lo que se puede regresar al scope padre.

Las tablas utilizan los hashes de los símbolos. Si hay colisiones utiliza una lista para agregar varios símbolos al mismo hash.

Los símbolos incluyen un nombre, un tipo y un contador de las veces que se accesa la variable. Tenemos una función lookup que busca recursivamente los símbolos, por lo tanto tenemos oversha-dowing.

1.4. Manejo de tipos

Para manejar los diferentes tipos que se pueden programar en cmm: *int*, *bool*, *double*, *string*[], *int*[], *bool*[], *double*[], se definieron constantes enteras para identificar los tipos.

1.5. Generación de código intermedio

Al momento de generar el código intermedio se consideraron muchas alternativas. En nuestro caso se utilizó el lenguaje intermedio de LLVM (`http://llvm.org/releases/3.6.2/docs/LangRef.html`).

1.5.1. LLVM

LLVM es una colección de herramientas que utilizan muchas empresas y lenguajes para compilar sus propios programas. Por ejemplo Apple utiliza LLVM como su infraestructura para compilar C, C++, Objective-C y Swift.

1.5.2. Runtime

Es necesario declarar un archivo que declare las implementaciones de las funciones incluidas en el lenguaje. Este archivo implementa las funciones **printInt**, **printDouble**, **printString** que utiliza internamente la función **print**. El archivo está escrito en el lenguaje intermedio.

1.6. Arquitectura

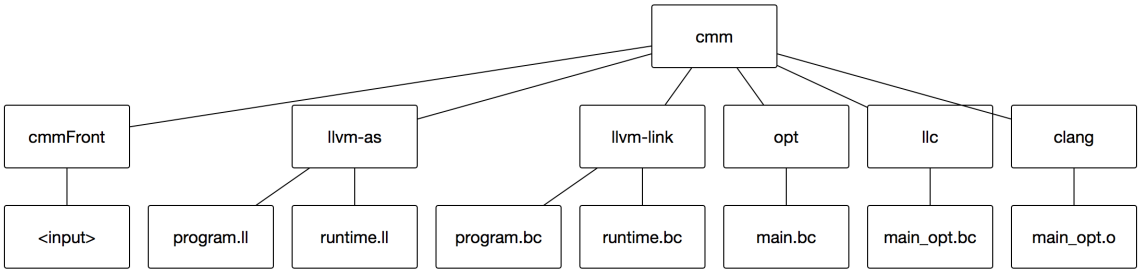

Figura 2: Arquitectura del compilador

Esta arquitectura corre en serie. Es decir el programa de hasta la izquierda recibe el archivo de input, corre el *front end* y genera el archivo **program.ll**. Las herramientas realizan lo siguiente:

1. **out/cmmFront**: Corre el *front end* que incluye el parser, el scanner y genera el código intermedio.
2. **llvm-as**: Convierte el código intermedio a bitcode.
3. **llvm-link**: *Linkea* los programas en bitcode.
4. **opt**: Optimiza el programa *linkeado* por **llvm-link**.
5. **llc**: Corre el *back end* y genera un archivo de objeto **.o**
6. **clang**: Corre el linker de otro compilador para *linkear* el **.o** con las *librerías* del sistema y generar un ejecutable.

1.7. Overshadowing de variables

Para manejar overshadowing de variables, generamos nombres diferentes en el archivo intermedio. Internamente en el *front end* utilizamos una estructura **t_symbol** que tiene dos cambios, **internalName** y **name**. **internalName** se asigna de acuerdo a las variables con el mismo nombre definidas en el bloque padre.

2. Manual de usuario

2.1. Compilación

Para compilar el programa tenemos un Makefile. Este archivo tiene las siguientes opciones:e

El comando **make** o **make help** muestran la documentación correspondiente de las opciones de make.

Una vez modificadas las variables, para compilar el programa utilizamos el comando: **make build**.

Para correr las pruebas del sistema utiliza el comando: **make test**.

Para limpiar los ejecutables utiliza el comando: **make clean**.

2.2. Corrimiento programa

Para compilar un programa utilizamos el comando **cmm**. El compilador tiene la siguiente ayuda:

```
usage: cmm [-h] [-o output] [--optimize]
           [--target {assembly,executable,intermediate}]
           source

Compiler for the cmm language. Currently uses a LLVM backend.

positional arguments:
  source                the source file to be compiled

optional arguments:
  -h, --help            show this help message and exit
  -o output             the output destination file. By default this value is
                        result.{target_extension}
  --optimize            optimize the generated intermediate code
  --target {assembly,executable,intermediate}
                        tells the compiler which is the target it should
                        generate
```