Compilador Kaskell

Álvaro García Tenorio [*] Miguel Pascual Domínguez[**]

29 de junio de 2018

Índice general

[*] alvgar14@ucm.es
[**] miguepas@ucm.es

Capítulo 1

Introducción

Comencemos primero explicando en que consiste nuestro lenguaje.

Un programa en código kaskell, consiste en 3 partes diferenciadas. Una primera parte que consiste en la declaración de registros. Nuestro lenguaje, permite la declaración de varios registros seguidos, que irán declarados uno detrás de otro, pueden estar separados por saltos de linea, espacios o sin separación. Además se permite la declaración de registros anidados, es decir, que uno de los campos de un registro, sea un registro previamente declarado.

La segunda parte consiste en los bloques. En nuestro lenguaje, los bloques están formados por la composición secuencial de instrucciones. Se permiten los bloques anidados, y las intrucciones pueden ser declaraciones de un array, un kinkeger (int) o un kool (bool), con o sin inicialización (asignación); asignaciones, khile (while), kor (for), if (if), kelse (else) o la llamada a una función.

Y por último, la tercera parte que consiste en la declaración de funciones. Las funciones pueden o no devolver un valor de tipo simple, es decir, un kinkeger o un kool, consideraremos que las funciones está formadas por dos partes, la parte de la cabeza que contiene los tipos de los argumentos, si los tiene, el identificador de la función y el tipo de retorno, si lo tiene; y la cola, que será los argumentos de tenerlos, el bloque de código y el valor de retorno, si lo tiene.

Capítulo 2

Estructura del compilador

Comentaremos ahora, la estructura del compilador, al igual del código del mismo. Para ello se comentará brevemente cada apartado del compilador.

2.1. Léxico

Comencemos con el léxico, en el léxico procesamos los identificadores de cualquier elemento del código, al igual que los símbolos especiales como "ƨ , y también las palabras reservadas.

Las palabras reservadas son: kif,kelse,khile,kor,kinkeger,kool, X (producto cartesiano), korr (or lógico), kand (and lógico), knot (not lógico), kod (operación modulo), kreturn (palabra reservada para el retorno de funciones), krue (true), kalse (false), y struk (registros).

Para poder analizar las palabras reservadas, nos ayudamos de un HashMap que tiene las palabras reservadas en el mismo, y al encontrar un identificador, comprobamos si es palabra reservada o no, viendo si está en el HashMap.

El resto de identificadores tienen una pequeña restricción de que solo pueden empezar por letras mayúsculas o minúsculas.

Y los comentarios de este lenguaje se escribe entre $. comentario .$.

2.2. Sintáctico

Continuemos con el análisis sintáctico del compilador.

Cup

El archivo Cup, consiste en la gramática de atributos que analiza la sintaxis del lenguaje, por tanto en vez de detallar la gramática, que esta ya presente en los archivos del compilador en el archivo "parser.cup". Vemos más conveniente, comentar informalmente la sintaxis del lenguaje.

Comencemos por las declaraciones. Hay 4 tipos de declaraciones: declaracion de tipo simple e identificador, array de tipo simple e identificador, tipo çomplejo"(tipo ya declarado, por ejemplo un registro) e identificador o array de tipo complejo e identificador. La sintaxis de estas cuatro declaraciones es: tipoSimple identificador, [tam 1º componente]...[tam i componente]...[tam n componente] tipoSimple identificador, IdentificadorTipoComplejo identificador, [tam 1º componente]...[tam i componente]...[tam n componente] IdentificadorTipoComplejo identificador.

Aclaremos que los tipos simples son kinkeger o kool.

Solo tenemos un tipo complejo que es la declaracion de los registros que es de la siguiente manera: identificador := struK declaraciones .

Veamos ahora las instrucciones: tenemos las instrucciones simples, complejas, y los bloques.

Un bloque se escribe de la siguiente manera: instrucciones .

Las instrucciones simples son las asignaciones, las declaraciones, y las declaraciones con asignaciones. Todas las instrucciones terminan con un ";". Las declaraciones ya las hemos aclarado antes, veamos como son las asignaciones y las declaraciones mixtas.

Las asignaciones son al igual que en los lenguajes tipo c++ o java, es decir identificador = expresión, identificador[i][j] = expresión, identRegistro.member = expresión.

Y las declaraciones mixtas, consisten en la declaración de un tipo simple y una asignación, es decir, tipoSimple identificador = expresión.

Veamos por último las instrucciones complejas. Estas son el if, el while y el for.

El if: kif (expresión) bloque o kif (expresión) bloque1 kelse bloque2.

El while: khile (expresión) bloque.

El for: kor (declaración mixta o asignación; expresión, asignación o expresión) bloque.

Las funciones se declaran de la siguiente manera:

La cabeza de la función: identificador : tipoArgumento1 X ... X tipoArgumentoN -¿ tipoRetorno. Recordemos que los argumentos y el retorno pueden no existir y por tanto se declararía como identificador :-¿.

Y la cola de la función: (identificador1, ...,identificadorN) -¿ bloque o bloque retorno.

Los bloques retorno son igual que los bloques normales con la salvedad de que tienen al final la instrucción kreturn indent/expresión/....

Además contamos con una gramática de expresiones aritméticas y lógicas, con las respectivas prioridades de operadores. La gramática de expresiones viene detalla en cup. Destacaremos que ciertas prioridades son distintas a lo habitual por ejemplo las de las comparaciones.

Árbol Sintáctico

Expliquemos ahora como esta organizado el árbol sintáctico. Como es de suponer tenemos varias clases java, que son lexer, cup, sym generadas por el léxico y cup que forman parte del compilador pero no son estrictamente pertenecientes al "árbol sintáctico", al igual que el main, que es la clase central del compilador.

Primero tenemos una clase global que es la que çontiene"todo el árbol que genera cup, que se trata de la clase programa, la cual contiene unos bloques, structs y funciones, además de la tabla de símbolos del programa, la cual tiene .ªlmacenados"los identificadores, con la ayuda de una interfaz medianamente importante que es Definition, para así de esta manera, poder saber en que ámbito del programa de se definen (funciones, structs o variables).

Como es lógico tenemos 3 clases que configuran lo que es cada parte del programa, veamos un poco cada una de ellas.

La clase function tiene la cabeza y la cola de la función como dos atributos de clase, ya que ambas son una clase por ellas mismas y estas 3 juntas forman el paquete de las funciones, cabe destacar que la cola, implementa la interfaz Definition, ya que como necesitamos saber en que ámbito esta definida la función, lo tenemos por la ayuda de la tabla de símbolos, aunque se podría haber hecho que function fuera la que lo implementara en vez de solo la cola, escogimos la cola por comodidad.

Veamos ahora el struct y el resto de los tipos, dado que struct forma parte del programa y consiste en la definición de un tipo nuevo, hemos considerado que StructType implemente

Definition también, y además extiende de Types. El resto de clases en el paquete types, son lo que su propio nombre indica, los arrays que extienden de Types, la clase Type, que engloba los tipos simples, y los complejos indirectamente al extender de ella, y el enumerado Types de los tipos simples.

También como es de esperar, una parte del árbol, esta formada por la gramática de expresiones que ya comentamos antes en la parte de cup. Tenemos una interfaz Expression, que nos sirve para englobar los distintos tipos de terminales con ciertas operaciones de control que tienen todos para poder determinar en que fila y columna están, además del tipo que tienen: los kinkegers en DummyInteger, los kools en DummyBoolean, los identificadores en Identifier y los arrays en ArrayIdentifier, los struk o registros en StructMember y por último las llamadas a funciones, pues en una expresión que devuelva un tipo, se puede poner en una expresión de ese tipo. Además tenemos dos enumerados que son los operadores binarios (BinaryOperatos) y unarios (UnaryOperators) que los "manejan"las clases BinaryExpression y UnaryExpression, respectivamente.

Vayamos ahora con la parte extensa del árbol, que son los bloques o instrucciones, pues al fin y al cabo, un bloque es una instrucción.

Por tanto hemos creado una interfaz llamada Statement que contiene los métodos que deben poseer todas las instrucciones, que es el chequeo de tipos y de identificadores. Con esto, consideramos que los bloques son una lista de staments, es decir, de instrucciones unas seguidas d otras. Y con ello el ReturnBlock, es una clase que extiende de Blocks que lo único que añade es la expresión de tipo simple del return de una función, por tanto dicho ReturnBlock solo estará en funciones.

Veamos ahora como son las instrucciones, de la interfaz Statement, de ella surgen dos clases, BasicStatement y ComplexStatement, una interfaz por extensión y una abstracta por implementación, respectivamente.

La interfaz BasicStatement, sirve para aclarar un poco la comprensión del código pues no añade nada más, y de ella heredaran las clases que implementarán las instrucciones simples, que son la asignación, la declaración de variables, y las mixtas (declaración y asignación). La asignación de registros o struk, se trata de una manera distinta y por tanto se usa una clase aparte llamada StructAssignment pero sigue siendo una extensión de la clase Assigment que es la de la asignación.

Por otro lado, la clase abstracta ComplexStatement, nos permite añadir un atributo que consiste en un bloque, con esto, basta darse cuenta que las instrucciones que pueden tener un bloque, son el while, el for, el if y el if else. Tenemos para el manejo del for, una clase auxiliar que se llama ForTuple, para manejar la tupla de la sintaxis del for ya mencionada en la parte de cup.

2.3. Identificadores

La parte de los identificadores, básicamente consiste en comprobar si el código tiene los identificadores correctos a lo largo del programa, es decir, si no hay identificadores usandose de variables sin declarar, o de variables declaradas después del uso.

La manera para comprobar esto, es con la ayuda de la tabla de símbolos que ya hemos mencionado anteriormente, para ello se procede de manera estructural, en la que cada estructura le pide su chequeo de identificadores a cada una de sus componentes más pequeñas, hasta que finalmente ella ya es una unidad mínima, la cual devuelve su correspondiente valor del chequeo de identificadores, que en el caso de ser un identificador, lo comprobará en la tabla, pero si es una constante, devolverá true.

2.4. Tipos

La parte de tipos, consiste en si toda instrucción esta bien "tipada", es decir, en una asignación, el tipo de la izquierda y de la derecha coinciden, en una expresión igual, etc..

Por tanto la manera de proceder en este caso, es completamente análoga a la de identificadores, pues consiste en proceder de la misma manera estructural, solo que en determinadas situaciones, por ejemplo en una expresión, hay que comprobar que ambos lados de la expresión tienen el mismo y además el tipo que soporte el operador asociado, y además ambas partes deben estar bien tipadas, por lo que sus componentes lo deben estar.

2.5. Generación Código de la Máquina-P

En el caso de generación de código consiste en generar un archivo de texto, con el código que acepta la máquina-P.

Como es de esperar, al igual que en las dos anteriores se procede de manera estructural, usando los esquemas de generación de código visto en clase. Para ello es necesario saber, el valor de la dirección de memoria, para ello, nos ayudamos de la tabla de símbolos y de la clase Identificador, que gracias a ella sabemos los marcos de definición de las variables pues cada identificador tiene un atributo que se llama Delta, que básicamente es lo que tiene, y por tanto sabemos en que punto están almacenadas. Además tenemos también funciones que nos ayudan con los números especiales de los arrays, y los dos números del tamaño de argumentos y de la pila de expresiones que son necesarios al generar el código de una función.

2.6. Errores

Los errores se han ido tratando en cada una de las etapas, lanzando un mensaje de error con la fila y la columna asociadas al lugar del error, sea sintáctico, el cuál lo indicará cup; o sean de tipos o de identificadores, que lo hacen nuestras clases del árbol.

2.7. Ejemplo de Código

Presentamos ahora un ejemplo de código completo sin errores, en el cuál observamos todo lo mencionado antes:

```
A := struK {
               kinkeger num;
               kool cond;
            }

B := struK {
             kinkeger sum;
             [10][5] kinkeger vec;
             }

{
    A a;
    B b;
```

```
    a.num=78;
    a.cond=kalse;
    b.sum=0;
    kor(kinkeger i=0; i<10; i++){
        kinkeger j=4;
        khile(j>-1){
            b.vec[i][j] = b.sum + a.num*j-i;
            j--;
        }
        a.num = f(a.num,a.cond) -5;
    }
}

f: kinkeger X kool -> kinkeger
   (a,b)|->{kinkeger d; kif(b==kalse){d=0;}kelse{d=a;}b = knot b; kreturn d;}
```