UNIVERSIDAD DE LA HABANA

Compilador de COOL

Amanda Marrero Santos - C411
Manuel Fernández Arias - C411
Loraine Monteagudo García - C411

1 USO DEL COMPILADOR

DETALLES SOBRE LAS OPCIONES DE LÍNEAS DE COMANDO, SI TIENE OPCIONES ADICIONALES...

2 ARQUITECTURA DEL COMPILADOR

UNA EXPLICACIÓN GENERAL DE LA ARQUITECTURA, EN CUÁNTOS MÓDULOS SE DIVIDE EL PROYECTO, CUANTAS FASES TIENE, QUÉ TIPO DE GRAMÁTICA SE UTILIZA Y EN GENERAL, COMO SE ORGANIZA EL PROYECTO. UNA BUENA IMAGEN SIEMPRE AYUDA. Para el desarrollo del compilador se usó PLY, que es una implementación de Python pura del constructor de compilación lex and yacc. Incluye soporte al parser LALR(1) así como herramientas para el análisis léxico de validación de entrada y para el reporte de errores.
El proceso de compilación se desarrolla en 4 fases:

1. Análisis sintáctico: se trata de la comprobación del programa fuente hasta su representación en un árbol de derivación. Incluye desde la definición de la gramática hasta la construcción del lexer y el parser.
2. Análisis semántico: consiste en la revisión de las sentencias aceptadas en la fase anterior mediante la validación de que los predicados semánticos se cumplan.
3. Generación de código: después de la validación del programa fuente se genera el código intermedio para la posterior creación del código de máquina.
4. Optimización: se busca tener un código lo más eficiente posible.

Hablar de las fases de compilación. Cada una de estas fases se dividen en módulos independientes, que se integran en el archivo principal `main.py`

2.1 Análisis sintáctico

En la fase de análisis sintáctico del compilador se definen aquellas reglas que podemos llamar "sintácticas": los métodos empiezan por un identificador, las funciones contienen una sola

instrucción, etc. Son aquellas reglas que determinan la forma de la instrucción. Para este conjunto de reglas existe un mecanismo formal que nos permite describirlas: las gramáticas libres del contexto. Justamente, la fase de análisis sintáctico se encarga de validar que los predicados sintácticos se cumplan.
El análisis sintáctico se divide en 2 fases: en una se realiza el análisis léxico, con la construcción de un lexer y en la otra se realiza el proceso de parsing, definiendo la gramática e implementado un parser para la construcción del Árbol de Sintaxis Abstracta (AST).

2.1.1 Análisis léxico

En esta fase se procesa el programa fuente de izquierda a derecha y se agrupa en componentes léxicos (*tokens*) que son secuencias de caracteres que tienen un significado. Todos los espacios en blanco, comentarios y demás información innecesaria se elimina del programa fuente. El lexer, por lo tanto, convierte una secuencia de caracteres (strings) en una secuencia de tokens. Después de determinar los caracteres especiales de COOL especificados en su documentación se deciden las propiedades necesarias para representar un token. Un token tiene un lexema, que no es más que el string que se hace corresponder al token y un tipo para agrupar los que tienen una característica similar. Este último puede ser igual o diferente al lexema: en las palabras reservadas sucede que su lexema es igual a su tipo, mientras que en los strings y en los números estos son diferentes. Mientras que un token puede tener infinitos lexemas, los distintos tipos son predeterminados. Además, para el reporte de errores se guarda la fila y la columna de cada token.
La construcción del lexer se realiza mediante las herramientas de PLY. Para su construcción es necesario la definición de una variable `tokens` que es una lista con los distintos tipos de tokens. Después se especifica cuales secuencias de caracteres le corresponderán a cada tipo de token, esto se especifica mediante expresiones regulares. Para cada tipo de token se definió una función mediante la convención de PLY de nombrar `t_tipo`, donde `tipo` es el tipo de token, y en el docstring la expresión regular que lo describe.

- Hablar del sistema interno de PLY, cómo construye un autómata finito determinista para formar el lexer. En general, hablar un poco más de cómo realiza el análisis PLY por detrás
- Hablar de las particularidades de los comentarios y el parseo de los strings

2.1.2 Parsing

El proceso de parsing consiste en analizar una secuencia de tokens y producir un árbol de derivación. Por lo tanto, se comprueba si lo obtenido en la fase anterior es sintácticamente correcto según la gramática del lenguaje.
El parser también se implementó mediante PLY, especificando la gramática y las acciones para cada producción. Para cada regla gramatical hay una función cuyo nombre empieza con `p_`. El docstring de la función contiene la forma de la producción, escrita en **EBNF** (Extended Backus-Naur Form). PLY usa los dos puntos (:) para separar la parte izquierda y la derecha de la producción gramatical. El símbolo del lado izquierdo de la primera función es considerado el símbolo inicial. El cuerpo de esa función contiene código que realiza la acción de esa producción.
En cada producción se construye un nodo del árbol de sintaxis abstracta, como se hace en Listing 1. El parámetro `p` de la función contiene los resultados de las acciones que se realizaron para parsear el lado derecho de la producción. Se puede indexar en `p` para acceder a estos resultados, empezando con `p[1]` para el primer símbolo de la parte derecha. Para especificar el resultado de la acción actual se accede a `p[0]`. Así, por ejemplo, en la producción `program : class_list` construimos el nodo `ProgramNode` conteniendo la lista de clases obtenida en `p[1]` y asignamos `ProgramNode` a `p[0]`

Listing 1: Función de una regla gramatical en PLY

```
def p_program(self, p):
    'program : class_list'
    p[0] = ProgramNode(p[1])
```

El procesador de parser de PLY procesa la gramática y genera un parser que usa el algoritmo de shift-reduce LALR(1), que es uno de los más usados en la actualidad. Aunque LALR(1) no puede manejar todas las gramáticas libres de contexto la gramática de COOL usada fue refactorizada para ser procesada por LALR(1) sin errores.
La gramática especificada en el manual de COOL fue reconstruida para eliminar cualquier ambigüedad y teniendo en cuenta la precedencia de los operadores presentes.
Para la construcción del árbol de derivación se definieron cada uno de los nodos. El objetivo de dicho árbol es describir la forma en que la cadena es generada por la gramática. Intuitivamente, esta información nos permite "entender" sin ambigüedad el significado de la cadena.

2.2 Análisis semántico

A parte de las reglas sintácticas mencionadas anteriormente para que el compilador determine que programas son válidas en el lenguaje tenemos reglas que no son del todo sintácticas: la consistencia en el uso de los tipos, que cierta función debe devolver un valor por todos los posibles caminos de ejecución. Estas reglas son consideradas "semánticas", porque de cierta forma nos indican cuál es el significado "real" del lenguaje. Mientras que para los predicados sintácticos podemos construir una gramática libre del contexto que los reconozcan los predicados semánticos no pueden ser descritos por este tipo de gramática, dado que estas relaciones son intrínsecamente dependientes del contexto.
El objetivo del análisis semántico es validar que los predicados semánticos se cumplan. Para esto construye estructuras que permitan reunir y validar información sobre los tipos para la posterior fase de generación de código.
El árbol de derivación es una estructura conveniente para ser explorada. Por lo tanto, el procedimiento para validar los predicados semánticos es recorrer cada nodo. La mayoría de las reglas semánticas nos hablan sobre las definiciones y uso de las variables y funciones, por lo que se hace necesario acceder a un "scope", donde están definidas las funciones y variables que se usan en dicho nodo. Además se usa un "contexto" para definir los distintos tipos que se construyen a lo largo del programa. En el mismo recorrido en que se valida las reglas semánticas se construyen estas estructuras auxiliares.
Para realizar los recorridos en el árbol de derivación se hace uso del patrón visitor. Este patrón nos permite abstraer el concepto de procesamiento de un nodo. Se crean distintas clases implementando este patrón, haciendo cada una de ellas una pasada en el árbol para implementar varias funciones.
En la primera pasada solamente recolectamos todos los tipos definidos. A este visitor, `TypeCollector` solamente le interesan los nodos `ProgramNode` y `ClassNode`. Su tarea consiste en crear el contexto y definir en este contexto todos los tipos que se encuentre.
Luego en `TypeBuilder` se construyen todo el contexto de métodos y atributos. En este caso nos interesa además los nodos `FuncDeclarationNode` y `AttrDeclarationNode`.
En el tercer recorrido se define el visitor `VarCollector` en el que se procede a la construcción de los scopes recolectando las variables definidas en el programa teniendo en cuenta la visibilidad de cada uno de ellos.
Por último, en `TypeChecker` se verifica la consistencia de tipos en todos los nodos del AST. Con el objetivo de detectar la mayor cantidad de errores en cada corrida se toman acciones como definir `ErrorType` para expresar un tipo que presentó algún error semántico.

2.3 Generación de código

- Hablar del objetivo de la generación de código.
- Introducir y justificar la necesidad de la generación de código intermedio.

2.3.1 Código Intermedio

- Hablar de la estructura del código intermedio.
- Explicar el visitor que se utiliza para generarlo.

2.3.2 Código de Máquina

- Describir el código Mips que se genera.
- Explicar como se llegó a implementar con un visitor.

2.4 Optimización

- No tengo ni idea de qué poner aquí.

3 Problemas técnicos

Detalles sobre cualquier problema teórico o técnico interesante que haya necesitado resolver de forma particular.

- Hablar de los comentarios y los strings en el lexer.
- Hablar de la recuperación de errores en la gramática de PLY.
- Hablar de cómo se implementaron los tipos *built-in*.