UNIVERSIDAD EUROPEA DE MADRID

ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

Grado en Ingeniería informática

Compiladores y lenguajes formales

Compilador para python

Vicente David Mut Giraldo

Angel Esquinas Puig

Yago Iglesias Diaz

Índice

1. Introducción al problema..... 2

2. Solución propuesta.....2

3. Directivas de compilación.....3

4. Manual de usuario..... 4

5. Batería de pruebas..... 5

5.1 Tests con Programas Correctos.....5

5.2 Tests con Programas con Errores.....6

6. Comentarios y conclusiones..... 7

1. Introducción al problema

El objetivo de este proyecto es desarrollar un compilador para un subconjunto del lenguaje de programación Python. Este compilador deberá reconocer diversos tipos de tokens y estructuras de control, realizar análisis léxico y sintáctico, aplicar controles semánticos, y finalmente generar código intermedio y código ensamblador para ser ejecutado en el emulador MARS (MIPS Assembly and Runtime Simulator).

El compilador se desarrollará en tres hitos principales:

1. **Hito 1**: Definición de la gramática y procesamiento de un programa Python válido que incluya operaciones aritméticas, booleanas, un bucle, estructuras condicionales y comentarios. Se generará un fichero de salida con los resultados de la compilación.

2. **Hito 2**: Incorporación de controles semánticos y generación de código intermedio mediante Árboles de Sintaxis Abstracta (AST). Se añadirán funcionalidades como arrays de dos dimensiones y funciones.

3. **Hito 3**: Generación de código ensamblador MIPS y validación en MARS para todas las funcionalidades implementadas, especialmente arrays y funciones.

Para el análisis léxico se utilizará un archivo .flex (lexico_python.flex) y para el análisis sintáctico un archivo .y (gramatica_python.y). Además, se desarrollará un árbol de derivación (AST_python.h) y una tabla de símbolos (tabla_simbolos.h) para gestionar el proceso de compilación.

En las siguientes secciones, se detalla el trabajo realizado en cada hito, así como los desafíos y soluciones implementadas.

2. Solución propuesta

La solución se llevará a cabo en tres hitos:

Hito 1: Implementar un analizador léxico y sintáctico que reconozca y procese un programa válido de Python, con soporte para operaciones aritméticas, booleanas, un tipo de bucle, condiciones 'si', y comentarios. Al final de este hito, el compilador generará un fichero de salida indicando el resultado de la compilación.

Hito 2: Incorporar controles semánticos y generar código intermedio utilizando Árboles de Sintaxis Abstracta (ASA). El compilador también deberá realizar comprobaciones de tipos, verificación de nombres de procedimientos, y control de ámbito de variables. El código intermedio será almacenado en un fichero de salida.

Hito 3: Generar código ensamblador MIPS y validar su funcionamiento en el emulador MARS.

3. Directivas de compilación

Para compilar y ejecutar el compilador desarrollado, es necesario utilizar MinGW y la terminal MSYS. A continuación, se detallan los pasos necesarios para realizar la compilación:

1. **Instalación de MinGW y MSYS**

Asegurarse de tener MinGW instalado en tu sistema. Después, instalar MSYS, un entorno de terminal para Windows que permite ejecutar comandos Unix..

2. **Configuración del entorno**

Abrir la terminal MSYS y navegar hasta el directorio ejecutable donde se encuentra el archivo compilar.sh

3. **Compilación del proyecto**

Se ejecuta los siguientes comandos en la terminal MSYS:

bison -d -v ../src/gramatica_python.y
flex -o ../src/python.lex.c ../src/lexico_python.flex
gcc -o COMPILADO ../src/gramatica_python.tab.c ../src/python.lex.c
./COMPILADO ../pruebas/validas/input_aritmetico.py

O por el contrario ejecutar el archivo *compilar.sh* para ahorrarse ejecutar los anteriores comandos uno a uno

Descripción de los comandos

- `*bison -d -v ../src/gramatica_python.y* `: Este comando invoca Bison para procesar el archivo de gramática `gramatica_python.y`, generando dos archivos: `gramatica_python.tab.c` (código fuente en C de la gramática) y `gramatica_python.tab.h` (declaraciones de tokens y otros elementos).

- `*flex -o ../src/python.lex.c ../src/lexico_python.flex*`: Este comando invoca Flex para procesar el archivo léxico `lexico_python.flex`, generando el archivo `python.lex.c` que contiene el código fuente en C para el análisis léxico.

- `*gcc -o COMPILADO ../src/gramatica_python.tab.c ../src/python.lex.c*`: Este comando utiliza el compilador GCC para compilar los archivos generados por Bison y Flex (`gramatica_python.tab.c` y `python.lex.c`), creando un ejecutable llamado `COMPILADO`.

- `*./COMPILADO ../pruebas/validas/input_aritmetico.py*`: Este comando ejecuta el compilador recién creado (`COMPILADO`) con un archivo de código fuente en Python (`test.py`).

Siguiendo estos pasos, podrás compilar y ejecutar el compilador para el lenguaje Python.

4. Manual de usuario

Este manual guía al usuario en la instalación, compilación y ejecución del compilador de Python desarrollado con Flex y Bison. El compilador genera código ensamblador MIPS que se valida con el emulador MARS.

Para comenzar, asegúrate de tener instalados MinGW y MSYS en tu sistema. MinGW proporciona el entorno de desarrollo necesario, mientras que MSYS permite ejecutar comandos Unix en Windows. Puedes descargar MinGW desde su sitio oficial y asegurarte de incluir los paquetes `mingw32-base` y `mingw32-gcc-g++`. Si MSYS no se instala automáticamente, puedes obtenerlo desde el mismo sitio web.

Una vez instalados MinGW y MSYS, abre la terminal MSYS y navega al directorio donde has colocado los archivos del proyecto (compilar.sh). Para ello, usa el comando `cd` seguido de la ruta al directorio correspondiente. Por ejemplo, `cd /c/ruta/al/directorio/proyecto`.

Con la terminal ubicada en el directorio correcto, procede a ejecutar los comandos de compilación. Primero, procesa el archivo de gramática con Bison usando `*bison -d -v ../src/gramatica_python.y* `. Luego, procesa el archivo léxico con Flex mediante `*flex -o ../src/python.lex.c ../src/lexico_python.flex*`. Finalmente, compila los archivos generados con GCC con el comando `*gcc -o COMPILADO ../src/gramatica_python.tab.c ../src/python.lex.c*`.

Para ejecutar el compilador, asegúrate de tener un archivo con código fuente en Python en el directorio pruebas en la carpeta validas, por ejemplo, `*input_aritmetico*.py`. Ejecuta el compilador con el archivo como argumento usando `*./COMPILADO ../pruebas/validas/input_aritmetico.py*`. Esto generará el código ensamblador MIPS, que se mostrará en la terminal o se guardará en un archivo de salida.

Para validar el código ensamblador, abre MARS, carga el archivo generado y usa las herramientas del emulador para ensamblar y ejecutar el código. Verifica los resultados para asegurarte de que sean los esperados.

En caso de encontrar errores durante la compilación o ejecución, verifica que todos los archivos necesarios estén en el directorio correcto y que los comandos se ejecuten sin errores tipográficos. Revisa los mensajes de error en la terminal para obtener pistas sobre posibles problemas y asegúrate de que el archivo de código fuente en Python esté correctamente escrito.

5. Batería de pruebas

Para garantizar el correcto funcionamiento del compilador y validar su robustez, se llevará a cabo una batería de pruebas. Estas pruebas se dividen en dos categorías principales: pruebas con programas correctos y pruebas con programas que contienen errores. Esta sección detalla los procedimientos y ejemplos para ambas categorías.

5.1 Tests con Programas Correctos

Los tests con programas correctos aseguran que el compilador procese adecuadamente el código Python y genere el código ensamblador MIPS esperado. A continuación, se presentan algunos ejemplos de programas correctos que se utilizarán para estas pruebas:

Ejemplo 1: Operaciones Aritméticas Básicas

```
# archivo: validas/input_aritmetica.py

a = 6

b = 7

c = a * b

print(c)
```

Ejemplo 2: Operaciones Concatenación

```
# archivo: validas/input_cadenas.py

a = "Buenas " + "noches"

print(a)
```

Ejemplo 3: Condición "si"

```
# archivo: validas/input_if.py

a = 7

b = 8

if a < b:

    j = 8

    print(j)
```

```
end
```

Ejemplo 4: Condición "si" Completa

```
# archivo: validas/input_elif.py
a = 1.5
b = 2.5
c = 3.5
if a >= 3.5:
    print(a)
elif a >= 1.5:
    print(b)
else:
    print(c)
end
```

Ejemplo 5: Bucle "for"

```
# archivo: validas/input_for.py
suma = 4.0
for i in range (2):
    suma = suma - 2.0
end
suma = suma + 1.0
print(suma)
```

Estos programas se ejecutarán con el compilador para verificar que el código ensamblador MIPS generado sea correcto y que los resultados impresos sean los esperados.

5.2 Tests con Programas con Errores

Los tests con programas que contienen errores son esenciales para validar la capacidad del compilador de manejar errores sintácticos y semánticos de manera adecuada. A continuación, se presentan algunos ejemplos de programas con errores que se utilizarán para estas pruebas:

Ejemplo 1: Error Aritmetico

```
# archivo: errores/input_aritmetico.py

a = 6

b = 0

c = a / b

print(c)
```

Este programa tiene un error aritmético al dividir por cero.

Ejemplo 2: Error tipos

```
# archivo: errores/input_cadenas.py

a = "Hola"

b = 6

c = a + b

print(c)
```

Este programa tiene un error de tipos ya que no se puede sumar dos tipos diferentes.

Ejemplo 3: Error de Sintactico

```
# archivo: errores/input_elif.py

a = 7

b = 8

if a > b:

    j = 8

    print(j)

el a < b:
```

```
    j = 9
    print(j)
end
```

Este programa tiene un error sintactico porque elif no esta bien escrito.

Ejemplo 4: Error de Sintactico

```
# archivo: errores/input_if.py
a = 7
b = 8
if a > b
    j = 8
    print(j)
end
```

Este programa tiene un error sintactico porque if no posee los dos puntos.

Ejemplo 5: Error de Sintactico

```
# archivo: errores/input_for.py
suma = 4.0
for i in ra (2):
    suma = suma - 2.0
end
suma = suma + 1.0
print(suma)
```

Este programa tiene un error sintactico porque for no posee los range.

6. Comentarios y conclusiones

El desarrollo del compilador para Python ha permitido implementar y validar diversas funcionalidades básicas, como las operaciones aritméticas (suma, resta, multiplicación, división), estructuras condicionales ("si") y bucles ("for"). Estas funcionalidades han demostrado funcionar correctamente a través de nuestras pruebas.

Sin embargo, aún persisten algunos problemas en la gestión de errores semánticos más complejos y en el manejo avanzado de funciones y arrays. Aunque el compilador maneja adecuadamente muchos aspectos básicos del lenguaje, queda trabajo por hacer para asegurar un manejo robusto y completo de todas las características de Python.