Trabajo Práctico Especial

Autómatas, Teoría de Lenguajes y Compiladores

***Grupo**: Lóránt y Los Pibes*

***Integrantes**:* Diego Bruno, Santiago Swinnen, Lóránt Mikolás y Felipe Gorostiaga

***Fecha de entrega**:* 27/06/2018

***Titular**:* Juan Miguel Santos

***Adjuntos**:* Ana María Arias Roig y Rodrigo Ezequiel Ramele

Índice

Introducción

El presente informe tiene como objetivo describir el trabajo realizado durante la construcción de un compilador para el lenguaje definido por el grupo, en el marco del trabajo práctico especial de la materia Autómatas, Teoría de Lenguajes y Compiladores del Instituto Tecnológico de Buenos Aires.

Asimismo se explicarán las decisiones tomadas y dificultades encontradas a lo largo de la implementación.

Para el desarrollo del trabajo se utilizaron las herramientas Lex y Yacc tal como lo indica la Cátedra y el resultado del trabajo es un ejecutable que compila programas escritos en el lenguaje definido.

Descripción del lenguaje

Idea subyacente y objetivo del lenguaje

La premisa básica que guió el desarrollo del trabajo fue la de crear un lenguaje verborrágico, simple de leer y fácil de aprender para personas que necesiten resolver cuestiones básicas dentro del contexto de la programación sin demasiado conocimiento.

Para ello se buscó otorgarle al lenguaje la mayor cantidad posible de similitudes con la prosa cotidiana, aquella en la que se expresa el ser humano en su cotidianeidad, de forma tal que este lenguaje le sea familiar. Por lo tanto resultaba fundamental eliminar algunas de las limitaciones que tiene un lenguajes como c en cuanto a su sintaxis.

Se debe tener en cuenta que llamamos ***lylp*** al lenguaje construido y lo referenciamos con aquel nombre en las diferentes secciones del informe.

Consideraciones

Una primera consideración que vale la pena notar es que se realiza una diferenciación entre dos tipos de expresiones. Por un lado aquellas de tipo lógico, y por otro todas aquellas que no son de tipo lógico, por ejemplo una suma o una resta. La explicación de esta diferenciación es que en este tipo de lenguaje que se busca lograr, carece de sentido realizar una evaluación lógica de una expresión que no lo es. Por ejemplo, si esta diferenciación no se realizara, sería aceptable el siguiente fragmento de código:

```
number count is 7.
if 4 + 3
     start
```

```
            count is 9.
      end
```

Si bien puede sonar razonable para un programador experimentado, es mucho más intuitivo para un usuario poco experimentado el siguiente fragmento:

```
number count is 7.
if 4 + 3 is not equal to 0
      start
            count is 9.
      end
```

También vale la pena marcar que para el lenguaje buscado era suficiente realizar un "compilador en una sola pasada". Si bien cuenta con ciertas limitaciones (como que las variables deben estar previamente declaradas para ser usadas), este tipo de enfoque permite tiempos de compilación más rápidos y además nos resultó más práctico en su implementación. El programa se recurre una única vez y se genera el código c *on-the-fly*.

Para esto a medida que se recorre el programa se va construyendo un string en donde se concatenan mediante la función strcatN(...) en cada reducción los fragmentos correspondientes. Al llegar al símbolo especial se realiza la impresión del string obtenido. En la sección de dificultades encontradas se explica en mayor detalle este proceso.

Por último, queríamos aclarar que se eligió que la salida sea en lenguaje c. En las siguientes secciones se continuará explicando cómo logramos diferenciar el lenguaje ideado del que compila gcc. Los programas escritos en lylp son analizados sintácticamente y semánticamente, utilizando además una tabla de símbolos para almacenar los identificadores de variables.

La compilación final con gcc se realizó con los siguientes flags: -Wall -pedantic. Con esto se buscó garantizar que no se tenga ningún tipo de warning de gcc y que cualquier error de código sea "atrapado" por nuestro compilador y no gcc.

Desarrollo

Para empezar, se eliminaron símbolos comunes a casi todos los lenguajes de programación, que si bien contribuyen en gran medida a la fluidez de uso de los expertos, restan legibilidad, premisa fundamental del lenguaje. Tal es el caso de los siguientes símbolos que se reemplazaron por sus correspondientes expresiones escritas:

- **>** → is greater than
- **<** → is less than
- **==** → equals
- **>=** → is greater than or equal to
- **<=** → is less than or equal to
- **!=** → is not equal to
- **&&** → and
- **||** → or

Otras expresiones lingüísticas que pertenecen al lenguaje inglés, como el “if” y el “else”, lógicamente fueron conservadas.

Así como se utiliza ";" para finalizar una línea de código en el lenguaje C, en este caso se optó por hacerlo con ".". Con el fin de simular que una oración escrita en un texto.

En segundo lugar, el lenguaje permite el uso de dos tipos de datos básicos que llamamos number (enteros) y text (strings). Una de las particularidades del lenguaje es que permite la suma de los number y de los text. En el primer caso generando la suma aritmética y en el segundo la concatenación de los string sumados. Estos se se hizo para reforzar la facilidad del lenguaje.

En tercer lugar, buscamos hacer sencilla la impresión de expresiones por salida estándar por lo que el lenguaje cuenta con la palabra reservada **see.** A continuación presentamos un ejemplo. De esta forma se pueden imprimir tanto números como strings por salida estándar sin necesidad de indicar formato con *%d* o *%s* como en c:

```
text str is “salir a comer ”.
text str2 is “afuera a la noche”.
text str3 is str + str2.
see str3.
```

La salida obtenida es: “salir a comer afuera a la noche”.

En cuarto lugar, se permite comentarios de múltiples líneas. Para ellos se conservó la sintaxis de c por su clara visualización en un editor de texto y son eliminados en el lexx como se vio en los ejemplos de las clase teóricas.

En cuanto al manejo de errores de compilación, buscamos que el compilador rápidamente pueda advertir cuando hay algún problema en el programa. Por lo que se realizan verificaciones de tipo, de caracteres no reconocidos, redeclaración de variables, asignaciones inválidas, entre otras. Para ello se utilizó la función yyerror(...) que avisa al programador que hubo un error y le informa en qué línea de código sucedió.

En quinto lugar, el lenguaje permite una sencilla comparación de cadenas para las expresiones lógicas. Cuando el usuario ejecuta el siguiente código:

```
text str is “aaa ”.
text str2 is “ZZZZZZZ”.
if str2 is greater than or equal to str
    start
         see str.
    end
```

La salida obtenida es: “ZZZZZZZ”.
Es decir, se pueden comparar de las mismas formas que se compara un number.

Para terminar, vale la pena mencionar que se agregó una palabra reservada al lenguaje llamada **shout** que permite la impresión de expresiones en mayúsculas y con un signo de exclamación al final.

```
shout “el resultado  ” + “del partido”.
```

La salida obtenida es: “EL RESULTADO DEL PARTIDO!”.

Sintaxis y uso del lenguaje

A continuación se realizará una breve demostración del uso del lenguaje mediante ejemplos de las operaciones básicas.

- Declaración:

```
number mynumber.
```

- Definición:

```
mynumer is 3.
```

- Condición if-else:

```
if mynumber is equal to 3
     start
          mynumber is mynumber + 1.
     end
else
     start
          mynumber is mynumber - 1.
     end
```

- Loop con condición:

```
do
      start
            mynumber is mynumber + 1.
      end
while mynumber is lower than 9 and mynumber not equals 0
```

- Impresión en salida estándar con see:

```
see “welcome to lylp”.
```

- Impresión en salida estándar con shout:

```
shout “what a nice compiler”.
```

Descripción de la gramática

La gramática consta de diez símbolos no terminales que se describen a continuación:

- **PROGRAM**: es el símbolo inicial de la gramática. Su parte derecha está compuesta únicamente por el símbolo no terminal STEND.
- **BLOCK**: representa un bloque de código. Este bloque puede estar formado por una sola línea, un if, un if-else, un do-while o, más genéricamente, un conjunto de bloques.
- **STEND**: representa una abreviación de start-block-end, que describe su parte derecha, donde block es el bloque que se describió anteriormente.
- **LINE**: una línea puede estar compuesta por una asignación, una declaración, una definición o una impresión con la palabra reservada *see (o shout)*.
- **ASSIGNMENT**: es una asignación tal como se la conoce en los lenguajes más usados.

- **DECLARATION**: al igual que la asignación, representa lo mismo que una declaración en la mayor parte de los lenguajes.
- **DEFINITION**: es una declaración con asignación en la misma línea.
- **EXPRESSION**: se subdividen las expresiones en dos tipos. En primer lugar, las que se representan con este símbolo no terminal, aquellas de carácter no lógico, y en segundo lugar las de carácter lógico, que se describen a continuación.
- **LOGEXP**: se utiliza este símbolo de la gramática para aquellas expresiones booleanas, es decir aquellas que pueden tomar valor 1 o 0.
- **TERM**: un término puede ser una variable, un número o un string.

Dificultades durante el desarrollo

Uno de los primeros encontrados fue el manejo de variables y cómo identificar una variable previamente declarada. Para esto se utilizó una tabla de símbolos, que permite mantener un registro de las variables declaradas y su respectivo tipo. Los símbolos (variables) son insertados en la tabla al momento de una declaración/definición y si ya existía, se lanza un error por redeclaración de una variable.

En segundo lugar, al permitir expresiones que contengan operaciones de tipo text y number, sin tener conocimiento del tipo de la expresión (ya que se tenía un string de la expresión) resultaba imposible saber si se realizaba una asignación válida. Por ejemplo, si en *text var is expression* el tipo de *expression* era el correcto (es decir, text). Para solucionar esto se decidió que el tipo de los símbolos de la gramática sea Node* en yacc. En esta estructura de nodo se guarda un char* representando el string concatenado hasta ese nodo y un int que indica el tipo del símbolo (text/number). De esta manera, fácilmente se podía saber si las operaciones que realizaban eran válidas o no, basta con preguntar por el tipo del nodo.

En tercer lugar, al principio tuvimos algunos problemas para hacer funcionar la suma de strings: “str1 + str2” porque queríamos hacer la concatenación y no la suma de los punteros. Finalmente se resolvió insertando una función en c en el código de salida c llamada strcatP(...) que retorna la concatenación de strings.

Por último, al realizar el BNF para la gramática, por el agregado de producciones lambda en lugares que no era necesario se obtenían conflictos en yacc. Por lo tanto, se tuvo que analizar la gramática y se eliminaron las producciones lambda: si un símbolo de la derecha de la producción era anulable se agregó una producción derecha con el símbolo y otra sin el símbolo.

Instructivo de ejecución

Para ejecutar el compilador se deben seguir los siguientes pasos:

1. Ir al directorio Compiler-LyLP
2. Ejecutar el comando **make all**.
3. Luego si se quiere compilar un archivo para obtener el código en c:
   ./compiler < filename.lylp > filename.c
4. Por último compilar el resultado obtenido con gcc:
   gcc filename.c -o filename
5. Y para ejecutarlo: **./filename**

Para construir de manera más rápida los archivos ejecutables proveídos como test (siendo 5 con sin errores de compilación y dos con).

1. Repetir los pasos 1 y 2 explicados anteriormente.
2. Ejecutar **make test<i>** (donde <i> es el número del test que se quiere compilar: 1, 2, 3, 4, 5, 6 y 7.)
3. Ejecutar el archivo compilado: **./test<i>**

Para eliminar los ejecutables de los test y los archivos con los que se construyó el ejecutable del compilador (y.tab.c, y.tab.h, etc.). Ejecutar **make clean**.

Ante cualquier duda, problema o dificultad se puede contactar a cualquier integrante del grupo a través del correo institucional del ITBA.

Futuras extensiones

Debido a que para esta primera iteración del lenguaje no se lograron implementar todos los aspectos que nos habíamos propuesto en un comienzo exponemos en esta sección las futuras extensiones.

Considerando los objetivos establecidos para el lenguaje y el “target” de usuario que se presentó, resulta compatible con estas ideas la posibilidad de ofrecer facilidades para la escritura de notas, a modo de agenda o recordatorio.

Para llevar a cabo esta extensión, se considera la creación de un nuevo tipo de datos, llamado note, en la que se pueda agregar fácilmente datos de tipo texto así como también borrarlos. Cuya impresión en pantalla tiene un formato que simula un *post it* pegado en la pantalla.

También se buscarían añadir un tipo de dato float para manejar números de punto flotante.

Para finalizar sería deseable agregar la posibilidad de realizar funciones.

Conclusiones

En conclusión, la realización de este trabajo nos permitió entender y ejercitar el uso de herramientas como lexx y yacc. A su vez, permite poner en términos prácticos el análisis ascendente de un LALR(1) y ver el funcionamiento de un compilador.

También vale la pena marcar el aspecto creativo del trabajo que consiste en idear un nuevo lenguaje. Esta característica resultó muy interesante para el equipo.

Por último, aclaramos que algunas de las funcionalidades que se querían implementar, no se llegaron a finalizar en esta primera iteración del lenguaje (como el uso de notas). Por lo cual, estos *features* serían agregados en futuras iteraciones.

Referencias

A continuación se encuentra la bibliografía utilizada para realización del trabajo práctico especial. Vale la pena mencionar que se utilizaron tanto los contenidos teóricos explicados en la materia como los prácticos. Además se hizo uso del material proveído por la cátedra que se encuentra en campus.

1. https://www.tutorialspoint.com/c_standard_library/stdarg_h.htm
2. https://github.com/faturita/YetAnotherCompilerClass
3. https://github.com/faturita/LlvmBasicCompiler
4. http://dinosaur.compilertools.net/yacc/
5. http://www.gnu.org/software/bison/manual/html_node/Union-Decl.html