Instituto Tecnológico de Buenos Aires

72.39 - Autómatas, Teoría de Lenguajes y Compiladores

PE-Lang

Autores:
Matias Enrique Pavan

Legajo:
58296

Noviembre 2023

Índice

1. Introducción

PELang es un lenguaje de programación desarrollado con Bison y Flex para analizar ejecutables (PE) del sistema operativo Windows de una manera simplificada.

1.1. Funcionalidades

Dentro de las funcionalidades que ofrece el lenguaje se encuentra la capacidad de:

- Extraer información asociada al encabezado del ejecutable (file header)
- Extraer información asociada al encabezado opcional del ejecutable (optional header)
- Extraer información acerca de las secciones del ejecutable (.text, .bss, etc)
- Extraer información acerca de las funciones que importa el ejecutable (imports)
- Extraer información acerca de las funciones que exporta el ejecutable (exports)
- Hacer uso de operadores relacionales como $<$, $>$, , $\neq$, $\geq$, $\leq$
- Hacer uso de estructuras de control básicas de tipo IF-ELSEIF-ELSE, FOR y WHILE
- Hacer uso de un método para imprimir un reporte de todo lo previamente mencionado

2. Desarrollo y dificultades

2.1. Frontend

Para realizar el análisis léxico y sintactico del lenguaje se utilizo Flex y Bison. Inicialmente las cadenas de texto (strings), se definían mediante una expresión regular:

```
\"[^\"]+\"{ return StringPatternAction(yytext, yyleng); }
```

Listing 1: Vieja manera de detectar un string

Luego fueron debidamente re-implementadas para que constituyan un contexto al igual que los comentarios lo que permite consumir la cadena de texto sin tener que estar preocupado por casos especiales que tengan que poder ser capturados por la expresión regular que una defina:

```
"\"" { BEGIN(STRING); }
<STRING>[^"]* { return ... }
<STRING>"\"" { BEGIN(INITIAL); }
```

Listing 2: Forma actual de detectar un string

El lenguaje inicialmente contaba con miembros de estructuras *hardcodeados* en la gramática:

```
"directory_entries" { return ... }
"directory_entries.imports" { return ...; }
"directory_entries.exports" { return ...; }
"dll" { return ...; }
"imports" { return ...; }
"exports" { return ...; }
"name" { return ...; }
"address" { return ...; }
"sections" { return ...; }
"virtual_size" { return ...; }
"virtual_address" { return ...; }
"optional_header" { return ...; }
"magic" { return ...; }
"optional_header.magic" { return ...; }}
```

Listing 3: Vieja forma de detectar un miembro de estructura

El cual traía implícito un problema que, si bien es rígida y estricta, lo cual simplifica el trabajo a realizar en el *backend*, me aumentaba considerablemente la gramática y su complejidad.

Esto se terminó solucionando eliminando todas estas definiciones y quedandome únicamente con la definición del identificador, para la cual luego en el *backend* verifico que el miembro no sólo sea válido, si no que tambien pertenezca al tipo de dato correspondiente:

```
member: IDENTIFIER DOT IDENTIFIER                { $$ = ...; }
       | member DOT IDENTIFIER                   { $$ = ...; }
       ;
```

Listing 4: Actual forma de detectar un miembro de estructura

Previo a comenzar con la implementación del backend, se tuvieron que solucionar **156 errores de reducción**, causados mayoritariamente por omitir el agregado de reglas de precedencia para ciertos operadores:

```
%left ADD SUB
%left MUL DIV
%left EQUAL NOT_EQUAL LESS_THAN LESS_THAN_OR_EQUAL GREATER_THAN
   GREATER_THAN_OR_EQUAL
%left AND OR NOT
```

Listing 5: Actual precedencia de operadores

Si bien también hubo errores del tipo *reduce/reduce*, estos mismos fueron solucionados cuando se re-implemento la producción *member* previamente mencionada. Se quiso, además de agregar el FOREACH, agregar un FOR clásico al estilo C, pero por cuestiones de tiempo y entrega no se terminó de implementar la parte de la generación de código (todo lo referido al AST, type-checking y demás ya está implementado y totalmente funcional, es más, si uno descomenta las reglas de Bison para el FOR clásico, el programa va a generar el FOR en Python).

Para ver el desglose de correcciones realizadas con respecto a la primera entrega, puede referirse al siguiente *issue*: https://github.com/Reversive/pe-lang/issues/19

2.2. Backend

2.2.1. Symbol Table

Se implemento una tabla de símbolos ya que al realizar el *transpiling* a Python, uno necesita verificar que todas las variables utilizadas dentro del código fueron debidamente declaradas para no obtener ningún error al ejecutar el código generado. Para ello, se definió la siguiente interfaz:

```
typedef struct {
    char* id;
    Type type;
    union {
        int intVal;
        char *strVal;
        Expression *expVal;
    } value;
    int uid;
} SymbolEntry;

typedef struct {
    SymbolEntry* entries;
    int size;
} SymbolTable;

SymbolEntry* SE_New(char* id, Type type);
SymbolTable* ST_New();
void ST_Free(SymbolTable* table);
int ST_AddSymbol(SymbolTable* table, SymbolEntry* entry);
SymbolEntry* ST_GetSymbol(SymbolTable* table, char* id);
int ST_SymbolExists(SymbolTable* table, char* id);
```

Listing 6: Definicion tabla de simbolos

Donde se tiene un *array* de *SymbolEntry*'s y se tiene un seguimiento del tamaño para poder ir agregando y nuevos símbolos al arreglo. En particular, una *SymbolEntry* va a guardar el tipo de dato, el nombre del identificador, un valor de inicialización (si es que lo tiene), y un identificador único que se usa más adelante en la fase de generación de código para evitar colisionar identificadores en un programa de mi lenguaje que pueden ser palabras reservadas en Python.

2.2.2. Scopes

Para los Scopes se implementó una especie de "pila":

```
typedef struct {
    SymbolTable** scopes;
    int size;
    int current;
} Context;
Context* CX_New();
void CX_Free(Context* context);
void CX_AddScope(Context* context);
SymbolEntry* CX_AddSymbol(Context* context, SymbolEntry* entry);
SymbolEntry* CX_GetSymbol(Context* context, char* id);
int CX_SymbolExists(Context* context, char* id);
void CX_MoveDown(Context* context);
```

Listing 7: Definicion pila de scopes

En donde se tiene un *array* de tablas de símbolos y se tiene seguimiento del tamaño y de la posición actual del cursor de la pila para poder agregar y buscar símbolos. Cabe destacar que esta interfaz expone una funcionalidad totalmente agnóstica del scope actual en el que me encuentro ya que es Flex quien se encarga de manipular el cursor en base a si detecta el token para indicar que se está ingresando o saliendo hacia/de un nuevo scope:

```
token OpenBracePatternAction() {
       LogDebug("[Flex] OpenBracePatternAction: '{'.");
       CX_AddScope(state.context);
       yylval.token = OPEN_BRACE;
       return OPEN_BRACE;
}

token CloseBracePatternAction() {
       LogDebug("[Flex] CloseBracePatternAction: '}'.");
       CX_MoveDown(state.context);
       yylval.token = CLOSE_BRACE;
       return CLOSE_BRACE;
}
```

Listing 8: Flex scopes

2.2.3. Type checking y Code Generation

Para ambos se siguieron los lineamientos provistos por la cátedra, es decir, teniendo el árbol AST ya generado, se procedió a crear un metodo **generate** por cada nodo distinto del árbol. Se decidió implementar type checking ya que, junto con la tabla de símbolos, son componentes que, sin ellos, muchos programas generados de salida no podrían ser interpretados (es decir, no fallarían por runtime, si no que lo harían mucho antes por algún error en la sintaxis).

Se realizaron verificaciones de tipos para los siguientes casos:

- Asignación de una expresión a un identificador (los tipos deben coincidir)
- Verificar que una estructura dentro del FOREACH es iterable (por ejemplo, *PESection section in pe.sections*, no sólo debería pasar que una es un subtipo del otro, si no que también el tipo a la derecha debe ser iterable)
- Operaciones entre expresiones
- Verificar parámetros para ciertas funciones built-in (por ejemplo, peopen solo puede recibir una string por parámetro, o en efecto, un identificador que es de tipo string)
- Verificar que una propiedad (member) pertenece a una instancia de una estructura (por ejemplo, verificar que si tenemos *PEFile pe* y *PESections sections = pe.sections*, efectivamente *sections* sea una propiedad del tipo *PEFile*)
- Verificar si una variable no esta inicializada y se intenta utilizarla luego para validar una expresion

Se decidió en **no implementar coerción de tipos** ya que no es necesario, explícitamente porque solo existen dos tipos nativos que son *int* y *string*, los cuales no se puede aplicar coerción entre ellos.

Con respecto a la generación de código, como fue antes mencionado, se realizó un *transpiling* a Python. Para ello, se tuvo que tener en cuenta la indentación a la hora de generar el código, para ello se diseño una librería para poder volcar el código generado:

```
typedef struct {
    FILE* file;
    int indentationLevel;
} OutputBuilder;

OutputBuilder* OB_New(char* file);
void OB_Write(OutputBuilder* builder, char* format, ...);
void OB_WriteWT(OutputBuilder* builder, char* format, ...);
void OB_IncreaseLevel(OutputBuilder* builder);
void OB_DecreaseLevel(OutputBuilder* builder);
void OB_Free(OutputBuilder* builder);
```

En particular, para el runtime, se utiliza el módulo pefile que simplifica el proceso de *parsing* para trabajar con este tipo de archivos. En particular, como prólogo al programa generado, se utiliza un archivo llamado *template.py* que tiene los siguientes contenidos:

```
from pe_analyzer import PEAnalyzer
import json
```

Para ver el desglose de las tareas realizadas con respecto al *backend*, puede referirse al siguiente *issue*: https://github.com/Reversive/pe-lang/issues/8

3. Futuras modificaciones

A continuación se listan las modificaciones posibles para el lenguaje:

- Existen muchos métodos en Bison que toman los mismos parámetros (por ejemplo, varias *expressions*) que podrían ser reducidos a un solo método que reciba un parámetro extra donde se especifique el tipo de expresión a la que fue reducida.
- Aumentar la usabilidad del compilador dando mensajes de error mas precisos y descriptivos.
- Utilizar una *hash table* para la tabla de símbolos en vez de un *array*, lo cual ayudaría a reducir la complejidad temporal de búsqueda de un simbolo de $\mathcal{O}(n)$ a $\mathcal{O}(1)$.
- Permitir la generación de un HTML para visualizar la estructura del EXE.
- Realizar un profiling del compilador para identificar las zonas de mayor complejidad computacional con objetivo de eficientizar la generación del programa

4. Manual de instalación/uso

4.1. Instalación

Se precisan ciertos pre-requisitos instalados para podes compilar el proyecto:

- Bison v3.8.2
- CMake v3.24.1
- Flex v2.6.4
- GCC v11.1.0
- Make v4.3

Para compilar el proyecto por completo, ejecute los siguientes comandos en la raíz del repositorio (en Linux):

```
user@machine:pe-lang/$ chmod u+x --recursive script
user@machine:pe-lang/$ script/build.sh
```

Sin embargo, en un entorno de Microsoft Windows, debe ejecutar:

```
user@machine:pe-lang/$ script/build.bat
```

Luego, la solución generada *bin\Compiler.sln* debe abrirse con el IDE de Microsoft Visual Studio 2022. Los ejecutables que genera este sistema se colocan dentro de los directorios *bin\Debug* y *bin\Release*, según corresponda.

4.2. Uso

Para el uso del proyecto, debe tenerse pre-instalado los siguientes componentes:

- Python 3.9
- Pipenv

Una vez instalados, en linux, ejecutar el comando:

```
user@machine:pe-lang/$ script/start.sh program
```

En el caso de que se encuentre en Windows:

```
user@machine:pe-lang/$ script/start.bat program
```

Donde *program* es el archivo donde se especificara el código a ser generado por el compilador. Al ejecutar el compilador, se generara en la carpeta *output* un archivo llamado *generated.py*. A continuación, dentro de la carpeta *output* se debe ejecutar el siguiente comando para ejecutar el codigo generado:

```
user@machine:pe-lang/output$ pipenv run python generated.py
```

A continuación detallo un programa de prueba:

```
main {
    PEFile pe = peopen("./binaries/setup.exe");
    PEHeader header = pe.file_header;
    print "PE Header:";
    print "\tMachine: ", header.machine;
    print "\tNumber of sections: ", header.number_of_sections;
    print "\tTime date stamp: ", header.time_date_stamp;
    print "\tCharacteristics: ", header.characteristics;

    PEOptionalHeader optional_header = pe.optional_header;
    print "PE Optional Header:";
    print "\tAddress of entry point: ", optional_header.aoep;
    print "\tImage base: ", optional_header.image_base;
    print "\tSection alignment: ", optional_header.section_alignment;
    print "\tFile alignment: ", optional_header.file_alignment;
    print "\tSubsystem: ", optional_header.subsystem;

    PESections sections = pe.sections;
    print "PE Sections:";
    for (PESection section in sections) {
        print "\tName: ", section.name;
        print "\tVirtual size: ", section.virtual_size;
        print "\tVirtual address: ", section.virtual_address;
        print "\tSize of raw data: ", section.raw_data_size;
        print "\tCharacteristics: ", section.characteristics;
    }

    PEImports imports = pe.imports;
    print "PE Imports:";
    for (PEImport import in imports) {
        print "\tDll: ", import.dll;
        for (PEFunction importFunction in import.functions) {
            print "\t\tName: ", importFunction.name;
            print "\t\tAddress: ", importFunction.address;
        }
    }
    peclose(pe);
}
```

El binario *setup.exe* ya se encuentra en la carpeta *binaries*, el hash correspondiente al mismo es *402dabc189f5a7ba6f04df5aee8e73bc*