SUG - Compilador

Buchieri Giovanni
Vázquez Santiago
Guzman A. Uriel

Taller de Diseño de Software

Informe de Implementación

2024™

Introducción

En el desarrollo del proyecto en cuestión, compilador de mini-lenguaje, tuvimos que ordenar muchas ideas referidas a la implantación del mismo, tanto para la lectura del archivo (parte lexicográfica), como así también en la parte sintáctica (parser) y en el parte de la semántica (interprete/compilador); para ello optamos por mantener las siguientes convenciones:

Análisis Léxico

En la definición de tipos de datos, los cuales puede enteros o booleanos, se usan los términos *int* y *bool* para su representación, además, se cuenta con el tipo *void* para darle la posibilidad a una función de no devolver nada, la cual se podría tomar como un procedimiento.

Tipos de datos

Para la notación de los términos numéricos se emplea la idea de definición recursiva *{digito}+,* donde digito es un entero comprendido en el rango [0-9], estos van a trabajar con las operaciones arrítmicas clásicas suma (+), resta (-), producto (*), división (/) y modulo (%), su representación original simplifica la lectura y comprensión.

Para la notación de los valores booleanos se usan las palabras reservadas *true* y *false*, las cuales aparte de reflejar la idea, son útiles para la comprensión. Este tipo de dato, a su vez cuenta con las operaciones lógicas binarias de and (&&) y or (||), y con la operación unaria not (!), habiéndose elegido dichos símbolos por razones similares a las de los valores numéricos.

Identificadores

Para la notación de las variables o nombres de los procedimientos se emplea los identificadores (id), denotados de la forma recursiva *{letra}({letra}|{digito}) *,* donde letra es un carácter del código ASCII comprendido en el conjunto [a-z] ∪ [A-Z].

Operaciones de comparación

Para el manejo de las estructuras y flujos del programa, se definieron las operaciones: asignación (=), valida para ambos tipos de datos mencionados, tendrá el tipo de los valores que se apliquen sobre ella; igualdad (==), válida para ambos tipos, contendrá el valor booleano correspondiente a la evaluación correspondiente; mayor que (>) y menor que (<), ambos validos solo para los valores enteros, y al igual que la igualdad, contendrá el valor de su evaluación.

Estructuras

En cuanto a estructura se refiere, como se mencionó anteriormente se cuenta con sentencias de control, estas se pueden clasificar en condicionales (*if_then_else*), iterativas (*while*) y lo que viene siendo un bloque principal (*Program*), el cual a su vez cuenta con una función *main,* la cual funciona como bloque de ejecución, la cual se define como una función y puede tener diferentes tipos de retorno, además de una sentencia de retorno (en caso de que el tipo no sea *void*) la cual denotamos *return*. Además de *main*, se definen las funciones en general, con la misma idea que *main*, con un *id* y tipo de retorno.

Complementos

Para la incorporación de elementos externos al archivo corriente, se optó por definir la palabra reservada *extern* como puente o direccionador para los mismos.

Comentarios

Para permitir realizar comentarios, anotaciones o aclaraciones de código, definimos los comentarios de una línea (//) y de varias líneas (/**/), dejando evidente su parecido al de otros lenguajes, lo que facilita su uso.

Delimitadores

Para una correcta lectura, se emplean ( ) como delimitadores, para saber que acción debe tener precedencia en las operaciones numéricas/booleanas y también, como encapsulante de sección de parámetros en una función. Otra cuestión a destacar es el uso de { } como delimitadores de bloque, útiles para saber dónde empieza y donde termina una secuencia de sentencias. Sin embargo, las llaves solo son empleadas en bloque, para diferenciar cada sentencia particular se emplea '*;*', y para la separación de parámetro se emplea ','.

En consecuencia, de lo mencionado, cabe destacar que puntos (.), espacios en blanco ( ) y saltos de línea (\n) se podrán emplear, pero siendo ignorados. Cualquier otro símbolo que esté mesclado con los previos o en caso de simple aparición, se producirá un error por ser una secuencia invalida para el lexer.

Para el posterior manejo de los símbolos, se devolverá un token asociado a cada uno de los mencionados anteriormente, los cuales serán trabajados y descriptos en el análisis sintáctico.

Análisis Sintáctico

Para la manipulación de los tokens rescatados en el análisis léxico, optamos que estos sean las etiquetas (nombre) que decoren los nodos de un árbol binario. En dicho nodo almacenamos un símbolo (estructura con los campos: identificador, valor, tipo, tamaño, etc.), y dos referencias, una para el hijo izquierdo y otra para el hijo derecho. De esta manera, al ir leyendo el archivo, se va creando el árbol sintáctico correspondiente, el cual empleamos en la evaluación durante el análisis semántico. De forma paralela, mientras se crea el árbol cada símbolo creado que corresponda a una variable o una función/procedimiento, se ira agregando a una tabla de símbolos.

El árbol sintáctico se basa en un árbol binario con dos referencias a sus nodos sucesores, mientras que la tabla de símbolos se creó a partir de la idea recursiva de usar una lista enlazada. La definición de dichas estructuras se realiza en los archivos *AST.c* y *symbol.c* presentes en el directorio *src*.

Para la creación del Árbol Sintáctico Abstracto (*AST*) y su manejo general, se emplearon los siguientes métodos:

```
// crear un AST tomando el símbolo, un nodo izquierdo y un nodo derecho
struct AST* createTree (Tsymbol* symbol, struct AST *l, struct AST *r);

// construye la table de símbolos una vez que el AST fue creado
void createTable(AST* ar);

// elimina todas las referencias de la tabla de símbolos asi como los nodos del AST
void elimArbol (AST* tree);

// toma el AST y lo recorre tipando y añadiendo los valores correspondientes a las
// evaluaciones a los nodos (usado en la creación del interprete)
void evaluate (AST* ar);

// plasman el AST en un archivo .dot para su visualización y verificar correcta creación
void showTreeDot (AST* tree,FILE* file);
void printDot (AST* tree, const char* filename);
```

Para la creación de la Tabla de Símbolos y realizar acciones sobre ella se emplean las siguientes funciones:

```
// crea un símbolo tomando el nombre del mismo, el tipo de valor que contiene,
// el tamaño que debe ocupar y la línea donde se encuentra en el código
struct Tsymbol * CreateSymbol (char *name, enum TYPES type, int size, int line);

// Devuelve un puntero a la entrada de la tabla de símbolos para la variable,
// devuelve NULL en caso contrario
struct Tsymbol *Lookup (char * name);

// busca una subtabla por su número de identificación, devuelve el puntero
// correspondiente o NULL en caso contrario
struct Tsymbol *LookupTable(int size);

// realiza la misma búsqueda que lookup pero una subtabla dentro de la principal
```

```
struct Tsymbol *LookupInTable(char * name,Tsymbol* symTabla);

// agrega un símbolo dado a la tabla, verifica que este ya no se encuentre allí
void Install (Tsymbol *symbol);

// elimina todas las referencias de la tabla de símbolos
void DeleteList ();

// muestral el contenido de la tabla, útil para verificar que se evaluó de forma correcta
void printable ();

// agrega una subtabla perteneciente a una función dada a la tabla de símbolos principal
void InstallTable(Tsymbol *symbol,Tsymbol *symTabla);
```

Para facilitar la búsqueda y acceso a las subtablas pertenecientes a las declaraciones de funciones, optamos, como se expresa en la descripción de la función lookupTable, por agregarle un valor numérico único a cada bloque de código. Esto lo hacemos de forma global en el momento de creación de cada bloque, lo que nos permite mayor facilidad de búsqueda y movilidad entre los distintos ámbitos del programa en el análisis semántico.

En cuanto a los tokens empleados, se encuentran categorizados según la función que cumplen, a diferencia de la etapa léxica donde se agruparon según su significado o tipo con el que trabajaba. Las distinción mencionada quedo de la siguiente manera: *declaraciones, tipos de datos, símbolos, y palabras reservadas.*

En *declaraciones* se encuentra solamente el token *ID*, siendo este usado para representar el nombre de la variable o función correspondiente. En *tipos de datos* se encuentran los tokens: *INT, TTRUE, TFALSE, TYPE_INT, TYPE_BOOL, TYPE_VOID*, siendo los que definirán los valores de los datos y tipos de retorno en las funciones. En *símbolos* se encuentran los tokens: *TMAS, TPOR, TMENOS, TDIVISION, TRESTO, ASIGNACION, TPAR_OP, TPAR_CL, TLLAVE_OP, TLLAVE_CL, OR, AND, NOT, MAYORQUE, MENORQUE* y *EQ*, usados en operaciones aritméticas como booleanas, además de estar presentes como delimitadores en los bloques de sentencias. Y en *palabras reservadas* se tienen los tokens: *PROGRAM, EXTERN, THEN, IF, ELSE, WHILE, RETURN* y *MAIN*, empleados como declaradores de bloques de código, importaciones y programa actual.

Cada token representa un tipo de evaluación diferente, pero en particular se tiene que los tokens: *ID, INT, TTRUE* y *TFALSE* son de tipo símbolo, con lo que al evaluarlos se deben tomar como dicho tipo de dato. Además, cada uno es empleado de manera particular según la regla gramatical que corresponda.

Para mantener coherencia y evitar conflictos de lectura, para algunos tokens definimos precedencia, esta viene dada de la siguiente manera: *OR, AND >> EQ >> MAYORQUE, MENORQUE >> TMAS, TMENOS >> TPOR, TDIVISION, TRESTO >> NOT*. Leyéndose de izquierda a derecha, el operador con más precedencia entre los mencionados es el *NOT* por ser operador unario, además este es el único que le definimos asociatividad a derecha, esto por cómo se debe evaluar.

Gramática

block: *TLLAVE_OP list_decls list_sents TLLAVE_CL*
| *TLLAVE_OP list_sents TLLAVE_CL*

prog: *PROGRAM TLLAVE_OP list_decls list_func main TLLAVE_CL*
| *PROGRAM TLLAVE_OP main TLLA VE_CL*
| *PROGRAM TLLAVE_OP list_func main TLLAVE_CL*
| *PROGRAM TLLAVE_OP list_decls main TLLAVE_CL*

main: *TYPE_BOOL MAIN TPAR_OP TPAR_CL TLLAVE_OP block TLLAVE_CL*
| *TYPE_INT MAIN TPAR_OP TPAR_CL TLLAVE_OP block TLLAVE_CL*
| *TYPE_VOID MAIN TPAR_OP TPAR_CL TLLAVE_OP block TLLAVE_CL*

declaracion: TYPE_INT ID ';' | TYPE_BOOL ID ';'

list_decls: ∅ | *list_decls declaracion*

asignacion: *ID ASIGNACION expr* ';'

sentencia: asignacion
| retorno
| if_else
| while
| call_func ';'

valor: *INT*
| *ID*
| *TMENOS INT*
| *TTRUE*
| *TFALSE*

expr: *valor*
| *call_func*
| *NOT expr*
| *TPAR_OP expr TPAR_CL*
| *expr TMAS expr*
| *expr TMENOS expr*
| *expr TPOR expr*
| *expr TDIVISION expr*
| *expr TRESTO expr*
| *expr MAYORQUE expr*
| *expr MENORQUE expr*
| *expr EQ expr*
| *expr AND expr*
| *expr OR expr*

list_sents: ∅ | *list_sents sentencia*

dec_parametro : *TYPE_INT ID* | *TYPE_BOOL ID*

parametros: ∅ | dec_parametro | dec_parametro ',' parametros

list_func: ∅ | list_func declare_funcion

argumento: ∅ | expr | argumento ',' expr

call_func : *ID TPAR_OP argumento TPAR_CL*

declare_funcion: *TYPE_INT ID TPAR_OP parametros TPAR_CL TLLAVE_OP block TLLAVE_CL*
| *TYPE_BOOL ID TPAR_OP parametros TPAR_CL TLLAVE_OP block TLLAVE_CL*
| *TYPE_VOID ID TPAR_OP parametros TPAR_CL TLLAVE_OP block TLLAVE_CL*
| *TYPE_INT ID TPAR_OP parametros TPAR_CL EXTERN ';'*
| *TYPE_BOOL ID TPAR_OP parametros TPAR_CL EXTERN ';'*
| *TYPE_VOID ID TPAR_OP parametros TPAR_CL EXTERN ';'*

retorno: *RETURN expr* ';'

if_else: *IF TPAR_OP expr TPAR_CL THEN TLLAVE_OP block TLLAVE_CL*
| *IF TPAR_OP expr TPAR_CL THEN TLLAVE_OP block TLLAVE_CL ELSE TLLAVE_OP block TLLAVE_CL*

while: *WHILE TPAR_OP expr TPAR_CL TLLAVE_OP block TLLAVE_CL*

Análisis Semántico

En cuanto a las decisiones sobre la semántica que se tomaron, se pueden distinguir las siguientes partes: aritmética (operaciones), booleana (condicional, negación, operaciones), retorno de valor, asignación y llamada de función. Éstas son vistas en la detección de errores y son las que definirán si es válida la entrada o no. Por otra parte, en cuanto a decisiones de diseño, para la creación de las estructuras y procedimientos empleados, se pueden tomar los puntos clave como una sola parte estructural.
En general se verifican los errores (hablando en estructura de árbol binario) para ambos nodos sucesores (si los contine), y se tiene un caso para cada posible caso de valor, pero a fines prácticos en esta sección solo se mencionarán los detalles de uso y valides de tipos para las sentencias permitidas, la definición de los casos se encuentra en el archivo Errors.c.

Aritmética

Para la evaluación aritmética se cumple que, si se está analizado un operador de suma, resta, multiplicación, división o modulo, ambas partes del mismo deben ser enteros, no se admitirán valores booleanos. Esto conlleva a que las operaciones mencionadas trabajen con variables enteras, funciones con retorno entero y constantes enteras, además de combinaciones de los tipos mencionados. Además de tomar valores concretos, admitimos que puedan tener resultados de otras operaciones, siempre y cuando sean de tipo entero.

Booleana (Operaciones)

Para la evaluación lógica se toma que, si se está analizado un operador de conjunción, disyunción, igualdad o diferencia (estos últimos 2 pueden ser usados con valores enteros), los valores tomados son de tipo variable booleana, función de retorno booleano, o constante booleana, además claro del uso recursivos de dichos operadores. Idénticamente al caso de operaciones aritméticas, se pueden usar resultados de otras operaciones lógicas.

Booleana (Condiciones)

Para la verificación de condiciones, se opto por asegurar que el único tipo de valor que se pueda reconocer sean los booleanos, en caso de que se pase un valor entero se producirá un error. Para estas se pueden emplear los mismo tipos de datos que en la operaciones, además de las operaciones mismas, puede usarse las operaciones de comparación mayor que y menor que, siempre que estas hayan sido usadas con valores enteros.

Booleana (Negación)

El operador de negación como se menciono anteriormente, puede contener cualquier tipo de operación, al igual que cualquier tipo de valor siempre que sea de tipo booleano, esta operación no admitirá datos enteros.

Retorno de valor

En simples términos, el retorno se puede tener o no en un bloque, este pude tomar valores enteros o booleanos, puede emplear expresiones que aritméticas o lógicas por igual, lo único que va a verificar es que el argumento que este posea, sea del mismo tipo que el bloque de función donde esta ubicado.

Asignación

Para la operación en cuestión se toma el mismo concepto que el retorno de poder emplear cualquier operación y dato en el lado derecho de la misma, siempre y cuando en el lado izquierdo se emplee una variable con el mismo tipo de valor.

Llamada a función

Para la invocación de una función los datos que esta toma deben ser en términos de cantidad y tipo, iguales, no se admiten más o menos parámetros de los que están definidos en el perfil de la misma, y cada argumento en su invocación debe ser del tipo correspondiente, porque al no usar equivalencia de tipos entre los booleanos y los enteros, cado uno de ellos representa un conjunto distinto. Estas pueden tomar como argumento otras funciones, siempre que cumplan con los mencionado anteriormente.

Estructurales

Para la estructura que presenta el programa por detrás, llevamos las dos estructuras mencionadas en el análisis sintáctico el árbol y la tabla de símbolos. Cada una aporta un enfoque diferente dependiendo si se trabaja en al detección de errores o en la generación de código.

Para la detección de los errores que pudieran surgir a partir de no respetar alguna de las cuestiones mencionadas previamente, se emplea la tabla de símbolos de manera tal que cuando un valor no corresponde con el tipo de la operación, este queda reflejado en su campo type de la tabla. Además de ser útil para los tipos, también nos permite verificar alcance. Cuando una variable o función se usa en alguna línea del código, su respectiva existencia debe ser chequeada en la tabla actual, en caso de no pertenecer al bloque de invocación se busca en un previo, de lo contrario se asume que no se ha declarado. Esta separación se logra al definir niveles para los bloques que se crean, para ello se emplea una variable global que lleva registro del nivel corriente.
Entre otras cuestiones que se llegaron a considerar, el manejo de la tabla se realiza de manera tal que la usamos como pila, entonces el uso principal es el de reconocedor sintáctico en la detección de errores, para la evolución en pasos posteriores directamente se emplea el árbol.

Código intermedio

Para la generación de código intermedio (código de tres direcciones) tomamos como base el árbol sintáctico, partimos de definir los operadores (+, -, *, &&, ||, etc.) como nodos que almacenan los valores resultantes, si bien esto ya lo hacíamos en la etapa en que se creó el intérprete, es decir, ya tenía esa finalidad, sin embargo, esta vez fue necesario hacerlo porque tomamos que cada operador representa una variable temporal, que almacena el resultado parcial. A partir de eso, creamos una serie de etiquetas y una estructura PseudoASM para el manejo de las mismas.

Etiquetas del código tres direcciones

T_ASIGN <asignado> <valor> <asignado> (asignación)
T_RETURN < > < > < > (indica el fin de una función)
T_IF < > < > < > (llamada a un condición if, no aparece explícitamente)
T_WHILE < > < > < > (ejecución de ciclo while, no aparece)
T_WF < > < > < > (fin del while)
T_OR <op1> <op2> < | | > (or lógico)
T_NOT <op1> < > <op1> (negación lógica)
T_AND <op1> <op2> < && > (and lógico)

T_IGUAL <op1> <op2> < == > (igualdad entre elementos)
T_MAYOR <op1> <op2> < '>' > (mayor que entre enteros)
T_MENOR <op1> <op2> < '<' > (menor que entre enteros)
T_SUM <op1> <op2> < + > (suma entre enteros)
T_RES <op1> <op2> < - > (resta entre enteros)
T_MOD <op1> <op2> < % > (modulo entre enteros)
T_PROD <op1> <op2> < * > (producto entre enteros)
T_DIV <op1> <op2> < / > (división entre enteros)
T_LABEL < > < > <nombreLabel> (etiqueta para saltos)
T_JUMP < > < > <nombreLabel> (salto sin condición)
T_IFF <condición> < > <nombreLabel> (salto por falso)
T_LOAD_PARAM < > < > <variable/constante/función> (carga de parámetros de una función)
T_FUNC < > < > <nombreFuncion> (inicio de una función)
T_END_FUN < > < > <nombreFuncion> (indica que se termina una función, no aparece explícitamente)
T_RET < > < > < > (indica que lo que se trata es una función)
T_CALL <nombreFuncion> < > < > (invocación de una función)
T_INFO < > < > < > (provee información de tipos)
T_REQUIRED_PARAM < > < > <param> (parámetro de la función que se carga)
T_GLOBAL < > < > <nombre> (declaración de variable global)

Estructura de PseudoASM

La estructura usada para esta etapa esta compuesta por cinco campos, *op1, op2* y *result* son de tipo *Tsymbol* ya que como se menciono anteriormente la idea era toamr el árbol como base y usar sus nodos tanto para la toma como para el almacenamiento de valores. Además, cuenta con un campo *tag* que contiene alguno de las etiquetas anteriores, útiles para el tratamiento de cada caso, y por último contine un campo *next* de tipo *PseudoASM**. Esto ultimo esta definido de esta manera, ya que el pseudo assembler fue pensado como una lista enlazada de instrucciones.

Código Objeto

Para la generación de código objeto (assembler) partimos agregando a la estructura *Tsymbol* un campo offset que nos permite no pisar valores usados previamente, además hará que cada nodo del árbol (variable/constante, operación o función) tenga su propio valor de desplazamiento, esto se hace en el análisis inicial dependiendo si se esta en una función u otra.

Entre las decisiones que debimos tomar en esta última etapa para mantener la consistencia de los programa y de código están:

- Cada función declarada que devuelva un valor debe contener una instrucción return obligatoriamente,
- El orden de declaraciones globales es: variables -> funciones,
- El orden del código dentro de un bloque es: declaraciones -> sentencias,
- Con respecto al assembler, optamos por asignar direcciones cada 16bits, es decir, si se declaran dos variables consecutivas sus offsets serán: -16 y -32 respectivamente.