Universidad Nacional de Río Cuarto

Facultad de Cs Exactas

Carrera de Licenciatura en Ciencias de la Computación

Taller de Diseño de Software

Profesor: Francisco Bavera (Pancho)

Integrantes:

Gonzalo Trepin
Esteban Galucci
Florencia Hernández

Informe de Proyecto de Compiladores

October 1, 2025

Contents

1 Análisis Léxico

1.1 División del trabajo

En esta etapa trabajamos en conjunto. La principal tarea consistió en reconocer los tokens del lenguaje (palabras reservadas, identificadores, literales, operadores, símbolos y comentarios) y plasmarlos en el archivo `flex.l`.

1.2 Decisiones de diseño y asunciones

El analizador léxico fue implementado con **Flex**, generando el archivo `lex.yy.c`. Se nos proporcionó un listado de tokens y reglas exactas. Sin embargo, la definición de ciertos elementos, como los identificadores, la dedujimos directamente de la gramática.

Por decisión de diseño, cada token reconocido se imprime en un archivo de salida (`lexout`) con fines de depuración, lo cual permitió seguir el flujo del análisis léxico de manera clara.

1.3 Diseño y decisiones clave

- Los identificadores se reconocen mediante la expresión:

  ```
  {letra}({letra}|{digito}|_)*
  ```

  asegurando que comiencen con una letra y luego puedan incluir letras, dígitos o guiones bajos.

- Los literales enteros se reconocen mediante:

  ```
  {digito}+
  ```

  convirtiendo el texto a entero con `atoi`.

- Los literales booleanos se mapean directamente a los valores `true` y `false`.

- Los comentarios de una línea se manejan con `//.*`, mientras que los comentarios multilínea se manejan con:

  ```
  "/*"([^*]|\n|(\*+[^*/]))*"*/
  ```

1.4 Detalles interesantes

Un detalle particular fue la necesidad de manejar comentarios multilínea. Si bien logramos implementar una expresión regular que funciona en la mayoría de los casos, aún presenta problemas en situaciones de **comentarios anidados**, esto todavia esta pendiente de solucionar, aunque no lo consifderamos algo critico, por lo que puede esperar.

2 Análisis Sintáctico (Parser)

2.1 Decisiones de diseño y asunciones

La gramática proporcionada contenía **ambigüedades** y algunas expresiones regulares. Por lo tanto, el primer paso fue **desambiguar la gramática y eliminar las expresiones regulares**, traduciendo estas reglas a una forma compatible con Bison sin perder el significado del lenguaje.

Para manejar operadores, se definieron precedencias usando `%left` y `%prec` para diferenciar operadores unarios y binarios.

2.2 Diseño y decisiones clave

- **Asociación de operadores:** Problemas de ambigüedad se resolvieron usando las directivas `%left` y `%right` de Bison.
- **Operador unario vs binario:** El operador - podía aparecer tanto como unario (negación) o binario (resta). Esto se resolvió usando la directiva `%prec UMINUS` en la regla correspondiente, diferenciando correctamente los contextos.
- **Conflictos shift/reduce por derivaciones vacías:** La gramática original permitía que ciertas reglas, como `var_decls` y `method_decls`, derivaran a λ (cadena vacía), provocando conflictos shift/reduce. Por ejemplo, la regla original:

  ```
  program : PROGRAM '{' var_decls method_decls '}';
  ```

  se transformó en:

  ```
  program
      : PROGRAM '{' var_decls method_decls '}'
      | PROGRAM '{' var_decls '}'
      | PROGRAM '{' method_decls '}'
      | PROGRAM '{' '}'
      ;
  ```

  Esto elimino la necesidad de tener la regla vacia en `var_decls` y `method_decls` , cubriendo todos los casos posibles y eliminando los conflictos.

2.3 Detalles de implementación interesantes

- Uso de `%prec UMINUS` para diferenciar operadores unarios y binarios.
- Expansión de reglas para eliminar derivaciones vacías y prevenir conflictos shift/reduce.

2.4 Problemas conocidos

Hasta el momento, el parser reconoce correctamente todos los constructos del lenguaje. No se han detectado conflictos sintácticos pendientes.

3 Construcción del Árbol de Sintaxis Abstracta (AST)

Para la construcción del AST, reutilizamos la definición que habíamos hecho en el pre-proyecto extendiendo la estructura existente para adaptarla al lenguaje completo.

3.1 Decisiones de diseño y asunciones

Se agregaron nuevos tipos de nodos que no estaban presentes en el pre-proyecto, ya que el lenguaje actual los requería: `NODE_UNOP`, `NODE_WHILE`, `NODE_IF`, `NODE_PROG`, `NODE_CALL` y `NODE_PARAM`.

Cada nodo del AST contiene:

- **Tipo de nodo (`NodeType`):** indica la categoría sintáctica (por ejemplo, entero, booleano, asignación, binop, función, etc.).
- **Campo `info`:** almacena información relevante, como nombre de variable o función, valor de literales enteros (`ival`) o booleanos (`bval`), y tipo de evaluación (`eval_type`). El `eval_type` permite determinar el tipo de retorno de funciones o de expresiones, y será útil en el futuro chequeo de tipos.

Actualmente, todos los nodos reservan espacio para la misma información, aunque muchos nodos no lo usan. Esto podría optimizarse en el futuro definiendo distintos tipos de `info` según el nodo.

3.2 Diseño y decisiones clave

- Cada nodo puede tener un **hijo izquierdo**, un **hijo derecho** y un **nodo `next`** que apunta al hermano derecho. Esto permite representar listas de elementos (por ejemplo, listas de sentencias, parámetros o argumentos) de manera sencilla.
- Se construye el AST de manera incremental durante el parseo, usando las acciones de Bison para crear nodos y conectarlos.
- Para nodos de funciones y variables, se inicializa `eval_type` según el tipo declarado (`integer`, `bool`, `void`).
- Para operadores binarios (`BINOP`), el tipo se determinará una vez que se conozcan los tipos de los hijos. Esto permitirá validar operaciones y asignar correctamente `eval_type` al nodo padre.
- Se mantuvo una estructura de árbol flexible que rompe ligeramente el concepto de árbol binario, pero simplifica el manejo de listas y recorridos.

3.3 Detalles de implementación interesantes

- El nodo `next` facilita recorrer listas de nodos, por ejemplo para iterar sobre parámetros de funciones o sentencias dentro de un bloque.
- Cada acción en Bison llama a `make_node` para crear nodos con la información correspondiente y conectarlos según la estructura del AST.
- Literales enteros y booleanos se crean directamente con su valor y tipo asociado (`TYPE_INT`, `TYPE_BOOL`).

4 Tabla de Símbolos y Análisis Semántico

En esta etapa, se realizo la construcción de la tabla de símbolos y el etiquetado del AST, para la tabla de simbolos se implemento la estructura y se construye recorriendo el arbol.

4.1 Diseño y decisiones clave

- La tabla de símbolos (`SymTab`) se implementa como una estructura anidada con puntero a `parent` y un nivel de `scope`. Menor nivel indica mayor jerarquía, de modo que el nivel 0 corresponde al scope global.

- Cada vez que se encuentra un nodo `BLOCK` en el AST, se abre un nuevo scope en la tabla de símbolos. cuando este bloque termina. se "desapila" toda la infomracion recolectada en ese nivel. esto tiene la desventaja que la tabla se construye y se destruye en el mismo momento de crearse. pero para la utilidad que le damos este funcionamiento es optimo. los niveles nos permiten manejar variables locales y globales de manera jerárquica, siguiendo el concepto de pila de scopes.

- Cada nodo `ID` o `CALL` en el AST se "etiqueta" (*labeling*) con su información correspondiente en la tabla de símbolos ya que ambas estructuras almacenan el mismo struct INFO. Es decir, el campo `info` del nodo apunta a la misma estructura `Info` que se encuentra en la tabla.

- Gracias a la jerarquía de niveles, siempre se toma la última declaración válida del símbolo en el scope más cercano. Por ejemplo:

```
integer x = 7;
bool test() {
    integer x = 6;
    return x == 7;
}
```

  Al evaluar `x == 7` dentro de `test()`, el `ID` se liga al `x` local cuyo valor es 6 ya que la otra declaracion de x se encuentra un nivel mas atras.

- Se tomó la decisión de que los parámetros de una función pertenezcan al mismo nivel que el bloque de la función. Para esto se distingue en `Info` si un nodo representa una función(ya que por el funcinamiento previo los parametros quedaban al mismo nivel que la declaracion de la funcion y no al nivel de su bloque), se almacena una lista de parámetros y el nivel de scope inicial es -1 hasta etiquetar el AST.

4.2 Estructura de información

La información almacenada tanto en la tabla de símbolos como en los nodos del AST está definida por la siguiente estructura:

```
typedef struct Params{
    char* param_name;
    TypeInfo param_type;
    struct Params *next;
} Params;
```

```
typedef struct Info {
    char* name;
    int ival;
    int bval;
    int scope;
    char* op;
    TypeInfo eval_type;
    int is_function;
    Params *params;
} Info;
```

como se ve el struct INFO sufrio cambios, los cuales tambien afectan al ast. si bien el funcionamiento no se ve afectado. a nivel arquitectural es una decision bastante pobre ya que se continua gastando memoria en informacion inutil. la mayoria de los campos se usan solo en casos especificos.

4.3 Detalles de implementación interesantes

- La tabla funciona conceptualmente como una pila: los scopes anidados permiten encontrar primero la declaración más cercana de una variable o función.
- El etiquetado del AST vincula cada nodo `ID` o `CALL` a la información de la tabla de símbolos correspondiente, lo que facilita el chequeo de tipos y otras verificaciones semánticas posteriores.
- Se implementan verificaciones clásicas de una tabla de símbolos: no se permite redefinir variables o funciones en el mismo scope, y se detecta uso de variables no declaradas.

5 Chequeo de Tipos (Typecheck)

Una vez creada la tabla de símbolos y etiquetado completamente el AST, se realiza el *typecheck* recorriendo el árbol y propagando los tipos hacia arriba. Cada nodo del AST contiene un campo `info->eval_type` que almacena su tipo, lo cual permite verificar la consistencia de tipos de manera eficiente.

5.1 Recorrido del AST

El *typecheck* se realiza mediante un recorrido completo del AST, siguiendo los nodos hijos (`left`, `right`) y hermanos (`next`). Según el tipo de nodo, se aplican las reglas de tipo correspondientes.

5.2 Chequeos realizados

- **Operaciones binarias (`NODE_BINOP`):**
  - Los operandos deben tener el mismo tipo.
  - Operadores de comparación (`==`, `<`, `>`) producen tipo `bool`.
  - Operadores aritméticos (`+`, `-`, `*`, `/`, `%`) requieren operandos enteros y producen tipo `int`.

  - Operadores lógicos (`&&`, `||`) requieren operandos booleanos y producen tipo `bool`.

- **Operaciones unarias** (`NODE_UNOP`):
  - El operador - requiere un entero.
  - El operador `!` requiere un booleano.
  - El tipo del nodo se hereda del hijo.

- **Asignaciones** (`NODE_ASSIGN`):
  - El tipo de la variable debe coincidir con el tipo de la expresión asignada.

- **Declaraciones de variables** (`NODE_VAR_DECL`):
  - Se chequea el tipo de la expresión de inicialización.

- **Retornos** (`NODE_RETURN`):
  - Si no hay expresión se espera `void`.
  - Si hay expresión, se compara con el tipo de la función (`func_type`).

- **Funciones** (`NODE_FUNCTION`):
  - Se guarda el tipo de la función en `func_type`.
  - Se chequean los tipos de parámetros y del bloque de la función.

- **Bloques, condicionales y bucles** (`NODE_BLOCK`, `NODE_IF`, `NODE_WHILE`):
  - Se asegura que las expresiones de condición sean booleanas.

- **Llamadas a funciones** (`NODE_CALL`):
  - Se comparan los tipos de parámetros reales con los formales.
  - Se detecta exceso o falta de parámetros.

5.3 Manejo de errores

Cada discrepancia de tipos genera un mensaje de error indicando el tipo esperado y el tipo real. Se utiliza la variable global `type_check_error` para registrar si hubo errores durante el chequeo. El recorrido continúa para detectar múltiples errores antes de detener la ejecución. luego de creer terminado el analisis de tipos, detectamos varios errores mas. por ejemplo para una operacion booleana de == la logica que seguiamos en typecheck era simplemente, mirar que el tipo de sus operandos sean iguales. esto a priori deberia funcionar, pero el lenguaje aceptaba cosas del tipo `void f(){} void main(){bool test = f() == f();}` esto carece de sentido. otro error que pudimos encontrar fue que podiamos llamar a funciones no existentes. es decir era valido `integer x = 0; integer y = x();` esto era por un error en la busqueda en la tabla de simbolos, ya que simplemente buscaba el primer identificador que coincida en nombre sin vericar si este es funcion o no. estos errores fueron arreglados.

5.4 Uso de `eval_type`

Cada nodo del AST almacena su tipo en `info->eval_type`, lo que permite propagar el tipo desde los hijos hacia los padres y verificar asignaciones, retornos y llamadas a funciones.

5.5 Ventajas del diseño

- Integración directa con la tabla de símbolos: cada nodo `ID` o `CALL` ya está etiquetado con su `info`.
- Chequeo de tipos consistente con la jerarquía de scopes.
- Permite detectar errores de manera precisa y localizada.

5.6 Limitaciones y mejoras

- Los mensajes de error no incluyen la línea exacta.
- No se implementan coerciones de tipos; todo debe coincidir exactamente.
- Se podría optimizar separando el chequeo de expresiones y declaraciones en funciones independientes.