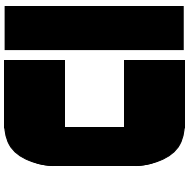


UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA CIVIL EN COMPUTACIÓN

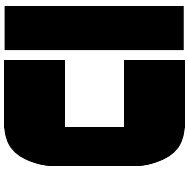
Implementación de una metodología para el diagnóstico y reparación automática de errores en el generador de analizadores sintácticos GNU Bison

CLAUDIO SAAVEDRA

Profesor Guía: CHRISTIAN F. ORELLANA

Memoria para optar al título de
Ingeniero Civil en Computación

Curicó – Chile
Marzo, 2008



UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA CIVIL EN COMPUTACIÓN

Implementación de una metodología para el diagnóstico y reparación automática de errores en el generador de analizadores sintácticos GNU Bison

CLAUDIO SAAVEDRA

Profesor Guía: CHRISTIAN F. ORELLANA

Profesor Informante: FEDERICO MEZA

Profesor Informante: JORGE PÉREZ

Memoria para optar al título de
Ingeniero Civil en Computación

Curicó – Chile
Marzo, 2008

En memoria de Luis Armando Saavedra Aguilera (1915–2002)

RESUMEN

Bison es un generador de analizadores sintácticos que genera, entre otros, analizadores LALR en el lenguaje C. El manejo de errores sintácticos en los analizadores generados por Bison es bastante pobre, ya que, ante la presencia de errores, los analizadores generados sólo pueden recuperarse en modo de pánico, esto es, descartando un fragmento de la entrada errónea y continuando el análisis en una ubicación subsecuente. Además, para poder obtener un diagnóstico de errores apropiado, los usuarios necesitan ensuciar las gramáticas de entrada con producciones de error especiales y, aún así, la corrección automática de errores no es posible.

En este trabajo, describimos los detalles de la implementación de una adaptación de la técnica de recuperación de errores por Burke y Fisher en Bison. Esta técnica intenta corregir los errores mediante la modificación de la entrada alrededor del punto de error, para luego continuar el análisis sintáctico, sin la necesidad de descartar código. Por esto, el diagnóstico de errores producido por esta técnica es de una calidad muy superior y la mayoría de los errores presentes son corregidos de manera automática, sin la necesidad de adaptar la gramática de entrada en absoluto.

ABSTRACT

Bison is a parser generator tool targeting, among others, LALR parsers in the C language. Error recovery in Bison is suboptimal since, in the presence of syntax errors, the parsers generated are only able to recover in panic mode, that is, discarding a fragment of the erroneous input, and resuming parsing in a subsequent location, discarding the syntactic details of the discarded fragment, as well. Moreover, in order to get proper error diagnosis, users need to clutter the input grammars with special error productions, and still, automated error correction is not possible.

In this work, we describe the details of the implementation in Bison of an adaptation of the error recovery technique by Burke and Fisher. This technique attempts to find a modification of the input that corrects the error, in order to resume parsing without the need to discard code. Error diagnosis produced by this technique are, therefore, of a much higher quality and the majority of the errors present are automatically corrected, without the need to modify the input grammar at all.

AGRADECIMIENTOS

Quisiera agradecer por sobretodo a mis padres y hermanas por su invaluable apoyo durante los largos años en Curicó y Dresde, y en especial por comprender mi necesidad de espacio y tranquilidad durante el tiempo que dediqué a este trabajo. Sin el constante aliento, la infinita paciencia, el apoyo incondicional y el entendimiento que me brindaron, no habría sido capaz de llegar a este punto. Gracias, de todo corazón.

Durante el curso de este trabajo, varias personas me brindaron valiosa orientación, incluso a la distancia. Primero que todo, gracias a Christian F. Orellana, por su constante orientación y apoyo. Quiero agradecer especialmente a Michael G. Burke, quien me dio importantes referencias sobre su técnica. Paul Eggert, Satya Kiran Popuri y otros desarrolladores de Bison respondieron siempre amablemente mis correos con consultas. Además, quisiera agradecer a las bibliotecas de la New York University y de la Universidad de Talca, en especial a Alejandra Garrido, por haber hecho posible que la tesis doctoral de Michael llegara a mis manos.

Quiero agradecer a Federico Meza por haber apoyado desde un comienzo mis intenciones de trabajar en este proyecto, y por haberme dado sus comentarios y sugerencias sobre algunos de los borradores iniciales de este trabajo. Agradezco además la buena voluntad de Jorge Pérez, quien accedió a formar parte de la comisión informante como Profesor invitado.

Quisiera agradecer además al proyecto GNOME, que me brindó por primera vez la oportunidad trabajar en un proyecto de software de primera clase. Por otro lado, estoy enormemente agradecido con el Deutscher Akademischer Austausch Dienst, por brindarme la posibilidad de estudiar en la Technische Universität Dresden durante un año y conocer ahí investigadores y profesores de primer nivel. Gracias a GNOME y a DAAD, obtuve el coraje necesario para enfrentarme sin miedo a un proyecto tan complejo como éste.

Y a mis amigos: gracias por estar siempre ahí acompañándome y dándome cada uno de esos bellos momentos que te dan fuerzas para continuar. Sin la belleza de las cosas simples que compartimos en el día a día, ningún esfuerzo habría valido la pena. *Gracias, Danke, Thank you.*

TABLA DE CONTENIDOS

Página

1. Introducción

Un compilador es una herramienta de software que permite la traducción del código de un programa en su forma humanamente leíble, conocida como *código fuente*, en una representación distinta que pueda ser analizada por una componente competente de un sistema computacional. Típicamente, esto implica la traducción del código fuente a una representación de máquina que pueda ser *ejecutada* por un computador de manera directa. Este proceso es conocido como *compilación*.

La compilación de código es un proceso bastante complejo. Por esto, se encuentra dividida en múltiples etapas. Inicialmente, el código fuente es dividido en cada una de las piezas que lo constituyen. Esta etapa se conoce como *análisis léxico*. A continuación, cada una de esas piezas, denominadas *tokens*, son analizadas y al código fuente se le impone una estructura gramatical. Esta etapa se conoce como *análisis sintáctico*. Luego, esta estructura gramatical es analizada para determinar la congruencia entre sus distintas partes, en una etapa conocida como *análisis semántico*. Posteriormente, se genera una *representación intermedia* que facilita la etapa final de *generación del código optimizado*. Este código optimizado es la salida del compilador y puede ser ejecutado por una máquina.

Este trabajo se concentra en la etapa de análisis sintáctico y, en particular, en dos conceptos importantes dentro de esta etapa: la utilización de *generadores de analizadores sintácticos* y el *tratamiento de errores sintácticos*. Un generador de analizadores sintácticos es una herramienta que permite facilitar la escritura de un compilador al traducir una representación sencilla de un lenguaje formal, e.g., una gramática en la forma de Backus-Naur, en la implementación de la maquinaria necesaria para poder generar y procesar el árbol sintáctico asociado a un fragmento de código perteneciente a dicho lenguaje, además de facilitar la tarea de detección de errores al usuario final.

GNU Bison es un generador de analizadores sintácticos que, a partir de la definición de una gramática libre de contexto en la forma de Backus-Naur, genera un analizador sintáctico LALR. *Bison* es software libre y parte del proyecto GNU [DS02].

1.1. Problema

En la actualidad, *Bison* sólo permite un tratamiento de errores de una forma bastante incómoda para los usuarios. Es necesario *ensuciar* la gramática de entrada, agregando *producciones de error*, para poder obtener algún grado de recuperación frente a la detección de errores. Más aún, esta recuperación es en sí limitada, ya que no permite que los errores sean corregidos de manera automática. Ante la presencia de errores sintácticos, *Bison* sólo intentará descartar la entrada errónea mediante una técnica en *modo de pánico*, de modo tal de continuar con el análisis sintáctico y permitir la detección de otros posibles errores.

A partir de los años '60 se han desarrollado distintas técnicas y metodologías para el tratamiento automático de errores en analizadores sintácticos. Estas técnicas intentan modificar la entrada cerca del punto de detección del error, de modo tal de encontrar una reparación que permita al analizador sintáctico, tanto entregar un diagnóstico del error, como continuar con el análisis, sin requerir mayor intervención del usuario.

En particular, Burke y Fisher propusieron, en el año 1987, una técnica que permite la reparación de errores en analizadores sintácticos, mediante la eliminación, inserción y/o modificación de tokens en la entrada del analizador, cuyas características lo hacen ideal para analizadores sintácticos usados en producción [BF87].

1.2. Objetivos

Este trabajo consiste, principalmente, en la implementación de la técnica de recuperación automática de errores sintácticos de Burke y Fisher en el generador de analizadores *GNU Bison*. Mediante esta implementación, pretendemos mejorar considerablemente la calidad del diagnóstico de errores y permitir una recuperación automática de los analizadores generados frente a la presencia de errores.

Hemos decidido implementar la técnica de Burke-Fisher en base a los méritos que presenta. Por un lado, la literatura sobre recuperación automática de errores

sintácticos la muestra como una de las técnicas más efectivas existentes. Por otro, es la técnica preferida por otros generadores de analizadores sintácticos, como *ML-Yacc*, un generador de analizadores para el lenguaje Standard ML.

1.3. Estructura del documento

Este trabajo está organizado de la siguiente manera. El Capítulo 2 presenta los conceptos teóricos relacionados con el análisis sintáctico. En el Capítulo 3 definimos los conceptos relacionados con la recuperación automática de errores sintácticos, discutimos algunas de las técnicas y metodologías más interesantes y presentamos en profundidad los conceptos relacionados con la técnica que implementamos. El Capítulo 4 presenta las distintas metodologías y herramientas que utilizamos a lo largo de nuestro trabajo. El Capítulo 5 revisa extensivamente los detalles de la implementación en *Bison* de la metodología escogida. En el Capítulo 6 analizamos los distintos resultados experimentales obtenidos y en el Capítulo 7 concluimos nuestro trabajo, presentando nuestras conclusiones.

2. Análisis Sintáctico

Una de las etapas centrales dentro de la compilación o traducción de código consiste en la identificación de su estructura sintáctica. Este proceso es conocido como *análisis sintáctico*, y para facilitarlo, durante los últimos 50 años se han desarrollado técnicas que permiten la generación automática de la maquinaria necesaria para realizar dicho análisis a partir de una gramática que especifique un lenguaje.

Las gramáticas pueden ser organizadas en distintas clases de acuerdo a la complejidad de la maquinaria necesaria para analizarlas y en la práctica se opta por utilizar clases de gramáticas que sean lo suficientemente expresivas como para representar lenguajes de interés práctico, pero, a la vez, lo suficientemente simples como para permitir un análisis eficiente.

Para apoyar la generación automática de los llamados *analizadores sintácticos* se han desarrollado distintas herramientas que permiten, a partir de la especificación formal de un lenguaje, generar automáticamente la implementación de una máquina que pueda analizar sintácticamente un texto con respecto a dicho lenguaje. *Yacc*, *GNU Bison*, *ANTLR* y *JavaCC* son algunos ejemplos de *generadores de compiladores* o *compiladores de compiladores*, que permiten automatizar la generación de analizadores sintácticos.

En este capítulo, definimos muy brevemente los conceptos relevantes al análisis sintáctico y repasamos las técnicas de análisis sintácticos más importantes, en las cuales se basa *Bison*, el generador de compiladores de interés en nuestro trabajo. Además, describimos brevemente *Bison*, sus características más importantes y el formato de sus archivos de entrada.

Para una descripción más detallada de las técnicas aquí descritas, recomendamos la revisión de los libros de Compiladores de Aho *et al.* [ALSU06] y de Appel y

Ginsburg [AG04]. Una comparación más detallada de las técnicas presentadas aquí y otras se encuentra en el libro sobre técnicas de análisis sintáctico de Grune y Jacobs [GJ90].

2.1. El problema del análisis sintáctico

En general, los lenguajes de programación cuentan con una naturaleza recursiva que hace imposible su definición por medio de expresiones regulares simples. Un ejemplo trivial de esto es un lenguaje compuesto sólo por pares de paréntesis balanceados. No es difícil demostrar que dicho lenguaje no es expresable por medio de una expresión regular [HMU06], lo que nos insta, en principio, a buscar una forma de expresarlo mediante una representación más poderosa.

Además, existe información inherente a un lenguaje recursivamente definido que es de interés. En particular, la estructura sintáctica de un lenguaje nos permite descomponer código en fragmentos para los cuales la generación o traducción sigue un patrón común. Ésta es una de las razones por las cuales la compilación de código se divide, en su etapa inicial, en *análisis léxico* y *análisis sintáctico*; durante el análisis léxico se identifica cada uno de los *tokens* que componen el código, tarea fácilmente realizable mediante un autómata finito determinístico; mientras que durante el análisis sintáctico se identifica y procesa la estructura de este código.

2.1.1. Preliminares

Un *alfabeto* es un conjunto finito no vacío de símbolos. Dado un alfabeto Σ cualquiera, la *palabra vacía* es una palabra que contiene cero símbolos, típicamente denotada por ε. Si Σ es un alfabeto y $a \in \Sigma$ un símbolo de Σ, entonces $a\varepsilon = \varepsilon a = a$ es una *palabra sobre el alfabeto* Σ. Si w es una palabra sobre Σ, entonces wa es también una palabra sobre Σ.

Sea Σ un alfabeto, $a \in \Sigma$ un símbolo, y w, x, y palabras sobre Σ. Entonces, definimos la *concatenación de w y x* como $wx = x$, si $w = \varepsilon$, o $wx = y(ax)$, si $w = ya$.

Sea Σ un alfabeto, $a \in \Sigma$, y w una palabra sobre Σ. El *largo de una palabra* w, denotado por $|w|$, se define como $|\varepsilon| = 0$, y $|wa| = |w| + 1$.

Sea Σ un alfabeto, w una palabra sobre Σ, y $k \in \mathbb{N}$. Entonces se define recursivamente la *k-ésima potencia de* Σ, como $\Sigma^0 = \{\varepsilon\}$, $\Sigma^{k+1} = \{wa \mid w \in \Sigma^k, a \in \Sigma\}$. El

conjunto de todas las palabras sobre un alfabeto Σ, Σ^* se define como $\Sigma^* = \bigcup_{k\geq 0} \Sigma^k$.

2.1.2. Gramáticas libres de contexto

La mayoría de los lenguajes de programación interesantes desde un punto de vista práctico están contenidos dentro de la clase de lenguajes conocidos como *lenguajes independientes de contexto*. Para estos lenguajes, existe una representación formal conocida como *gramática libre de contexto* [HMU06], a través de la cual queda definida completamente la estructura sintáctica del lenguaje. Formalmente, una gramática libre de contexto está definida por la tupla

$$G = (N, T, P, s)\,,$$

donde N es un conjunto de *variables* o *no terminales*, T es el conjunto de *terminales* o *tokens válidos* del lenguaje, $P \subseteq N \times (N \cup T)^*$ es un conjunto de *producciones* del lenguaje, y $s \in N$ es el *no terminal inicial*. Adicionalmente, definimos un *símbolo gramatical* como un símbolo $X \in N \cup T$.

Una *forma sentencial* α es una palabra compuesta por símbolos gramaticales, es decir, $\alpha \in (N \cup T)^*$. En particular, una forma sentencial compuesta sólo por terminales se conoce como *sentencia*.

Sea $G = (N, T, P, s)$ una gramática y $(A, \alpha) \in P$ una producción de G. Por comodidad, nos referimos a (A, α) como $A \to \alpha$.

Ejemplo 2.1 (gramática para un lenguaje). Considérese el lenguaje de todas las palabras compuestas únicamente por pares de paréntesis balanceados, descrito recursivamente como:

1. La palabra vacía representa una palabra compuesta trivialmente sólo por pares de paréntesis balanceados.

2. Si w_1 y w_2 son palabras compuestas por pares de paréntesis balanceados, entonces $(w_1)w_2$ también lo es.

3. Sólo las palabras construidas de acuerdo a los puntos 1 y 2 representan palabras compuestas por pares de paréntesis balanceados.

Sea L el lenguaje descrito. Una gramática que define exactamente a L es $G = (\{S\}, \{(,)\}, \{S \to \varepsilon, S \to (S)S\}, S)$.

A continuación, definimos formalmente los conceptos que nos permiten determinar el lenguaje generado a partir de una cierta gramática.

Derivaciones y lenguaje generado por una gramática

Sea $A \to \gamma$ una producción y α, β y γ formas sentenciales de una gramática. Entonces $\alpha A \beta \Rightarrow \alpha\gamma\beta$ es una *derivación en un paso*. Extendemos además la relación $\Rightarrow$ para definir una derivación en *cero o más pasos*, mediante el símbolo $\overset{*}{\Rightarrow}$. Esta relación se define inductivamente como:

1. $\alpha \overset{*}{\Rightarrow} \alpha$, para toda forma sentencial α.
2. Si $\alpha \overset{*}{\Rightarrow} \beta$ y $\beta \Rightarrow \gamma$, entonces $\alpha \overset{*}{\Rightarrow} \gamma$.

Una secuencia de derivaciones a partir de la variable inicial y que termine en una secuencia compuesta sólo por terminales, de modo tal que en cada derivación se reemplace el no terminal más a la izquierda de la forma sentencial a la izquierda, se conoce como *derivación más a la izquierda*. Análogamente, se define una *derivación más a la derecha* como la secuencia de derivaciones tal que en cada derivación se reemplaza el no terminal más a la derecha y que termina en una secuencia de terminales.

El *lenguaje $L(G)$ generado por una gramática G* está compuesto por todas las sentencias para las cuales existe una derivación desde el símbolo inicial hasta ellas. Es decir, si $G = (N, T, P, s)$, entonces $L(G) = \{w \in T^* \mid s \overset{*}{\Rightarrow} w\}$.

Árboles de derivación

Un *árbol de derivación* permite representar gráficamente una derivación particular. Se construye partiendo con el no terminal inicial en la raíz del árbol. Luego, por cada no terminal en el árbol, se agregan nodos hijos dependiendo de la derivación elegida para procesar el texto de entrada. Las hojas del árbol corresponden a los terminales.

Una gramática puede presentar más de un árbol de derivación para una determinada entrada. Estas gramáticas son conocidas como *ambiguas*. Esto significa que para ciertas entradas, un analizador sintáctico puede necesitar enfrentarse a la decisión de elegir entre dos o más alternativas de derivación, lo que implica que la estructura del

texto a procesar no está del todo definida. Idealmente, desearíamos contar sólo con gramáticas que no son ambiguas, pero en caso de existir ambigüedades en alguna gramática de interés, es posible definir reglas que ayuden a resolverlas.

Forma de Backus-Naur

A principios de los '60 Backus [Bac59] y Naur [BBG+60] propusieron un método para describir la gramática de dos lenguajes importantes en ese momento: Fortran y Algol, respectivamente. A partir del método que ambos propusieron, se desarrolló una técnica descriptiva conocida como *forma de Backus-Naur (BNF)*, la cual sintetiza la estructura de una gramática libre de contexto. A partir de entonces, BNF es la sintaxis preferida para describir lenguajes en contextos técnicos y, además, los generadores de compiladores, entre ellos *Bison*, la utilizan como formato de entrada para las gramáticas a procesar. En la sección 2.4.1 describimos la forma de BNF utilizada por *Bison*.

2.1.3. Técnicas de análisis sintáctico

En un comienzo, los diseñadores de compiladores realizaban implementaciones específicas de analizadores sintácticos directamente en el código del compilador, con la desventaja inherente de dificultar así la verificación de la equivalencia entre los lenguajes que se pretendía analizar y aquellos realmente analizados. Además, posibles modificaciones a la gramática requerían de complejas modificaciones al código, lo que dificultaba aún más la mantención de dichos proyectos.

A continuación, describimos algunas de las técnicas que se desarrollaron a partir de entonces para simplificar la construcción de analizadores sintácticos.

Análisis recursivo descendente

A principios de los '60 se desarrollaron técnicas conocidas como *recursivas descendentes*. Un *analizador recursivo descendente* está compuesto por un procedimiento por cada uno de los no terminales pertenecientes a su gramática. El análisis comienza al ejecutar el procedimiento asociado al no terminal inicial. Este procedimiento lee los primeros símbolos de la entrada e intenta hacerlos coincidir con uno de los lados derechos de las producciones cuyo terminal en el lado izquierdo corresponde al procedimiento en ejecución. De existir dicha correspondencia, el analizador llama

recursivamente a los procedimientos asociados a los no terminales en el lado derecho y el proceso continúa de esta manera.

Algunas gramáticas, en particular aquellas con recursividad por la izquierda, pueden causar un ciclo infinito en el análisis, dada la naturaleza recursiva de la técnica, lo que resta el interés en ellas.

Análisis descendente predictivo no recursivo

Un *analizador descendente predictivo no recursivo* está compuesto por una pila de símbolos gramaticales, una lista de *tokens de entrada* y una tabla de análisis. Mediante el uso de la pila, esta técnica emula la recursividad inherente al análisis.

Al principio del análisis, el no terminal inicial es apilado y el analizador apunta al primer token en la entrada. Luego, en cada iteración del análisis, si el símbolo en la parte superior de la pila y el token apuntado coinciden, ambos son descartados. De lo contrario, la tabla de análisis indica, a partir de ambos, la producción a utilizar. La recursividad se emula, entonces, al apilar los símbolos gramaticales al lado derecho de la producción elegida, en vez de llamar métodos recursivamente.

Las técnicas descendentes realizan el análisis partiendo por el no terminal inicial de la gramática asociada, prediciendo la estructura sintáctica en la entrada, para luego verificarla o descartarla, realizando luego una nueva predicción. En la medida que la estructura sintáctica predicha coincide con la existente en la entrada procesada, se profundiza en el árbol de derivación.

Desafortunadamente, estas técnicas fallan ante la presencia de una gramática ambigua o recursiva por la izquierda. Si bien gran parte de las gramáticas ambiguas o recursivas por la izquierda pueden ser transformadas para eliminar dichas propiedades, el costo asociado invita a la utilización de otras técnicas que sean capaces de lidiar con ellas.

Análisis ascendente no recursivo

Una clase de técnicas más poderosa está compuesta por aquellas conocidas como *bottom-up*. Estas técnicas intentan construir un árbol de análisis sintáctico partiendo por las hojas y los elementos más simples, para luego subir hasta llegar al no terminal inicial. Dentro de estas técnicas, aquéllas basadas en un programa con pila y una tabla de análisis son conocidas como *Left to Right-Rightmost derivation (LR)*, ya

que procesan la entrada de izquierda a derecha y producen una derivación más a la derecha.

En la sección 2.2 nos referimos en mayor profundidad a estas técnicas, ya que son las de mayor interés para el propósito de este trabajo.

2.2. Análisis sintáctico ascendente

Dentro de las técnicas más eficientes y poderosas existentes se encuentran las técnicas ascendentes. Las más importantes son las conocidas como técnicas LR(k), que procesan la entrada de izquierda a derecha y generan una derivación más a la derecha, utilizando k tokens de entrada para decidir el curso del análisis a seguir. Existe además una forma generalizada de analizadores sintácticos LR, conocida como GLR, capaz de manejar gramáticas ambiguas y no determinísticas y, aunque *Bison* puede generar este tipo de analizadores, no es de nuestro interés.

De especial interés es el caso particular de los analizadores LR con $k = 1$, ya que son capaces de manejar la gran parte de los lenguajes de interés y por lo tanto son los más usados en la práctica. Analizadores sintácticos con $k > 1$ son raramente usados, ya que las tablas en estos analizadores crecen de manera exagerada, sin garantizar que la clase de lenguajes que son capaces de analizar sea realmente más amplia. Incluso, se puede demostrar que para $k > 1$, todo analizador sintáctico LR(k) puede ser transformado en un analizador sintáctico LR($k - 1$) [GJ90].

En la práctica, los analizadores LR(1) son extremadamente grandes, incluso para gramáticas pequeñas. Por esto, existe además una simplificación de los analizadores sintácticos LR(1), conocida como LALR(1), la cual reduce considerablemente el tamaño del analizador al condensar la tabla de análisis.

En lo que sigue, describimos brevemente la maquinaria asociada a los analizadores LR(1) y LALR(1).

2.2.1. Analizadores LR(1)

Un *analizador LR(1)* cuenta con una pila de estados, una lista ordenada de tokens que aún no han sido procesados y una tabla de análisis. A partir del estado en la parte superior de la pila y del primer token en la entrada, el analizador efectúa una acción que le permita avanzar en el análisis, aceptar la entrada o bien reportar un error sintáctico.

Formalmente, podemos describir el estado de un analizador LR(1) mediante la secuencia

$$(s_0 \dots s_n t_0 \dots t_m)\,,$$

donde $s_0 \dots s_n$ representa la pila de estados, con s_n, el *estado actual*, en la parte superior de la pila, y $t_0 \dots t_m$ la entrada todavía sin procesar, con t_0, el *token actual*.

La tabla de análisis está compuesta por dos secciones: *ACTION* y *GOTO*. La sección *ACTION* permite definir la acción a ejecutar a partir del estado y del token actuales. Si $ACTION(s_n, t_0) =$ **desplazamiento** s, entonces el estado del analizador cambia a

$$(s_0 \dots s_n s t_1 \dots t_m)\,,$$

descartando el token t_0. Esta operación se conoce como *desplazamiento de* t_0 con *apilamiento de* s. Un desplazamiento de un token de entrada indica su aceptación como parte de la entrada y el apilamiento de un estado nos permite avanzar en el reconocimiento de una forma sentencial que eventualmente forma parte del lado derecho de una producción.

La sección *GOTO*, por su parte, nos permite determinar, a partir de un estado y un no terminal, el siguiente estado. Si $ACTION(s_n, t_0) =$ **reducción por** $A \to \gamma$, entonces el estado del analizador cambia a

$$(s_0 \dots s_{n-k} s t_0 \dots t_m)\,,$$

donde $|\gamma| = k$, y $s = GOTO(s_{n-k}, A)$. Esta operación se conoce como *reducción por la producción* $A \to \gamma$, y ocurre cuando hemos logrado coincidir una forma sentencial con γ, por lo que podemos reemplazarla por el no terminal A.

Los analizadores sintácticos LR(1) cuentan con la propiedad de *detección inmediata de errores sintácticos*, es decir, durante el procesamiento de entradas erróneas, un analizador LR(1) detectará un error sintáctico tan pronto como éste pueda ser detectado.

Una gramática para la cual existe un analizador LR(1) se conoce como *gramática LR(1)*.

2.2.2. Analizadores LALR(1)

Como ya mencionamos brevemente, una de las desventajas de los analizadores LR(1) es que la cantidad de estados y, por tanto, el tamaño de la tabla de análisis, puede ser extremadamente grande, incluso para una gramática simple. Por esto, un tipo de analizadores sintácticos conocidos como *Lookahead LR(1)*, o LALR(1) son de importancia en la práctica, ya que condensan la tabla de análisis reduciendo su tamaño considerablemente. Una gramática para la cual existe un analizador LALR(1) se conoce como *gramática LALR*.

Un *analizador LALR* se construye a partir de un analizador LR, al condensar estados obtenidos a partir de ítems con núcleos equivalentes. El algoritmo del análisis es equivalente al de la técnica LR(k) y las diferencias radican en la construcción de la tabla de análisis. La tabla de análisis construida mediante el algoritmo que genera un analizador LALR es conocida como tabla LALR.

No todas las gramáticas LR(1) son LALR. Dado que en un analizador LALR se condensan estados del analizador LR(1), durante el análisis pueden producirse situaciones en las cuales el algoritmo tenga que decidir qué reducción utilizar para reducir una forma sentencial. Esta situación puede ser detectada durante la construcción de una tabla LALR y, de ocurrir, entonces la gramática en cuestión no es LALR.

Una propiedad interesante de los analizadores LALR está relacionada con la detección de errores. Al intentar procesar una entrada errónea, un analizador sintáctico LALR no será capaz de detectar el error tan tempranamente como un analizador LR lo haría. Sin embargo, pese a que es posible que se realicen algunas reducciones extra, un analizador LALR siempre detectará un error presente en la entrada antes de reducir el siguiente símbolo.

2.3. Características adicionales de los analizadores sintácticos

Si bien, teóricamente basta con la pila de estados, la entrada y el algoritmo de control para poder tener un analizador sintáctico, en la práctica es deseable contar con ciertas características adicionales que hagan del analizador una maquinaria más útil. Con estas características, los analizadores sintácticos se convierten, no sólo en verificadores de correctitud sintáctica, sino que además pueden ser capaces de traducir la entrada, interpretarla y además, en caso de existir errores, brindar información

de diagnóstico de buena calidad a los usuarios.

2.3.1. Acciones semánticas

Para poder realizar trabajo adicional al análisis sintáctico con respecto a la entrada y a su estructura sintáctica, es necesario contar con cierta especificación adicional a la gramática. Esta especificación se conoce como *reglas o acciones semánticas* y está compuesta por una regla por cada producción de la gramática.

Dada una producción P y una acción semántica S asociada a P, el analizador sintáctico debe ejecutar S cada vez que realiza una reducción por P. La acción semántica incluye, por cada uno de los símbolos gramaticales en su producción, un *valor semántico*, que se puede utilizar como parámetro de la acción. Si S es un símbolo gramatical de P, entonces nos referimos a su valor semántico como $S.valor$.

Considérese un analizador sintáctico para la evaluación de expresiones algebraicas. Una de las producciones de la gramática asociada tendría la forma $E \rightarrow E_1 + E_2$. La acción semántica para esta producción tomaría los valores semánticos asociados a E_1 y a E_2, y los sumaría, asignándole el valor resultante al valor semántico de E. La acción semántica involucrada sería entonces $E.valor = E_1.valor + E_2.valor$.

En la práctica, estos valores semánticos son almacenados en una pila conocida como *pila de valores semánticos*. En la Figura 2.1, se puede apreciar un analizador sintáctico LALR(1) que cuenta con una pila de valores semánticos.

2.3.2. Ubicaciones

Por otra parte, en la práctica, los diseñadores de analizadores sintácticos pueden requerir de información sobre la ubicación en el archivo de entrada de cada uno de los tokens por procesar. Esta información, conocida como *ubicaciones*, permite entre otras cosas, una mejora de la calidad de los mensajes de advertencia y error durante la compilación.

En general, se puede considerar a las ubicaciones como un caso particular de acciones y valores semánticos. Típicamente, una ubicación es una tupla de cuatro valores que indican la columna inicial, columna final, fila inicial y fila final del token asociado. Durante una reducción, puede asignarse una ubicación para el no terminal en la cabeza de la producción, si se considera que su columna y fila inicial corresponden a la columna y fila inicial del primer símbolo del cuerpo y que su columna y

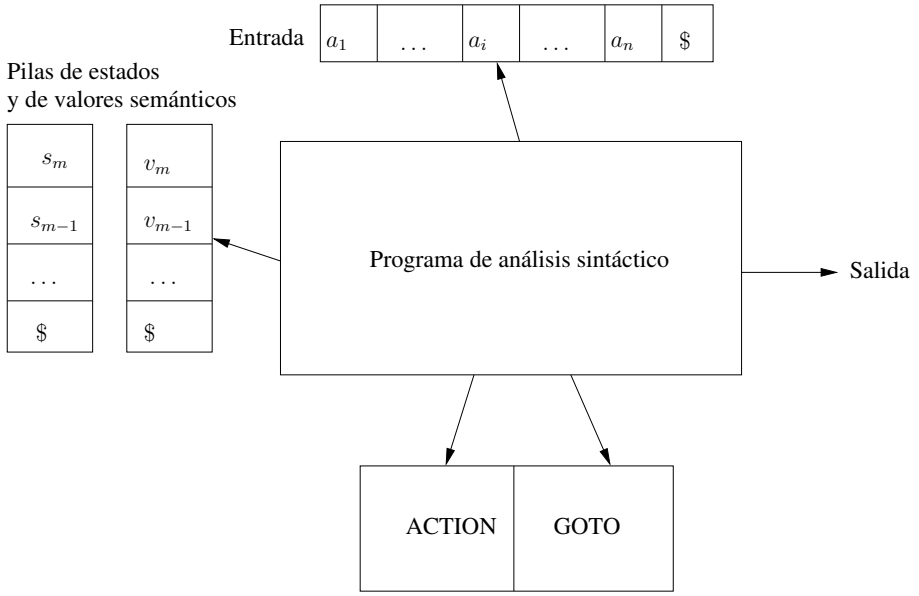


Figura 2.1: Analizador sintáctico LALR(1) con pila de valores semánticos

fila final corresponden a la columna y fila final del último símbolo del cuerpo.

Al igual que los valores semánticos, en la práctica, las ubicaciones se almacenan en una *pila de ubicaciones*.

2.4. Generadores de analizadores sintácticos

Un *generador de analizadores sintácticos* es una herramienta que permite generar, a partir de una descripción precisa y sencilla de una gramática, el código en algún lenguaje de alto nivel, que permita analizar sintácticamente archivos de entrada con respecto a dicha gramática. Considerando que la construcción de un analizador sintáctico es una tarea compleja pero bien conocida, estas herramientas facilitan bastante la construcción de traductores, intérpretes y compiladores. Por esto, estas herramientas también son conocidas bajo el nombre de *generadores de compiladores* o *compiladores de compiladores* [ALSU06].

En esta sección presentamos *Bison*, un generador de analizadores sintácticos LALR para sistemas GNU.

2.4.1. GNU Bison

Esta herramienta está basada en Yacc, el generador de analizadores sintácticos estándar en sistemas UNIX. *Bison* es software libre y es una herramienta común en un sistema GNU, como GNU/Linux, que ha sido utilizada para diseñar y construir variados compiladores y software utilizado en sistemas de producción [DS02, ALSU06].

Dada una gramática especificada en una sintaxis similar a BNF, *Bison* genera un módulo en el lenguaje C capaz de analizar sintácticamente el lenguaje generado por dicha gramática. El analizador sintáctico generado puede ser tanto LALR como GLR.

Un archivo de entrada para *Bison* está dividido en cuatro secciones: el *prólogo*, las *declaraciones*, las *reglas gramaticales* y el *epílogo*. El prólogo se utiliza para definir tipos y variables usadas en las acciones. Las declaraciones permiten definir los nombres de los terminales y no terminales de la gramática, además de definir ciertas propiedades como la precedencia de operadores y tipos semánticos de las reglas de derivación. Las reglas gramaticales especifican cada una de las producciones de la gramática. El epílogo es un espacio para que el usuario de la gramática especifique código adicional que pueda estar interesado en utilizar.

Ejemplo 2.2 (archivo de entrada para *Bison*). El Listado 2.1 presenta una gramática simple de entrada para *Bison* [Pop06a]. Podemos identificar el prólogo por encontrarse entre los símbolos `%{` y `%}` al comienzo del listado. A continuación se encuentran la declaración de los tokens existentes y la precedencia de los operadores. Para separar tanto las declaraciones como el epílogo de las reglas gramaticales, se utilizan los símbolos `%%`.

Al ser procesada con *Bison*, la entrada del Listado 2.1 produce un archivo fuente en el lenguaje C, que contiene un conjunto de funciones que permiten el análisis sintáctico de tokens con respecto a la gramática entregada. Dichos tokens son entregados mediante una función `yylex`, que puede ser escrita manualmente por el usuario o bien generada mediante una herramienta como *Flex* [PEM07].

En el capítulo 5 nos referimos extensamente al funcionamiento interno de los analizadores LALR generados por *Bison* y el trabajo necesario para adaptarlos a la metodología de recuperación de errores implementada.

Listado 2.1: Gramática para una calculadora minimalista en *Bison*.

```
%{
/* A simple calculator */
void yyerror(const char *s);
%}
%token        NUMBER
%token        OPERATOR_PLUS
%token        OPERATOR_MINUS
%token        OPERATOR_MULT
%token        OPERATOR_DIV
%token        LPAREN
%token        RPAREN
%left OPERATOR_PLUS OPERATOR_MINUS
%left OPERATOR_MULT OPERATOR_DIV
%%
exp: exp OPERATOR_PLUS exp
   | exp OPERATOR_MINUS exp
   | exp OPERATOR_MULT exp
   | exp OPERATOR_DIV exp
   | LPAREN exp RPAREN
   | NUMBER
;
%%
void
yyerror(const char *s)
{
  fprintf(stderr,"Syntax error: %s\n",s);
}
```

3. Recuperación automática de errores sintácticos

Un analizador sintáctico, ante la presencia de errores en la entrada, debe ser capaz de recuperarse de la mejor manera posible, de modo tal de continuar el análisis. Esta recuperación puede ser tan limitada como una recuperación *en modo de pánico*, que simplemente descarta el fragmento de código erróneo o lo suficientemente compleja como para determinar una modificación que corrija completamente el error. En cualquier caso, se espera que la recuperación efectúe un diagnóstico apropiado, que entregue al usuario información suficiente para comprender la naturaleza del error.

Una *técnica de recuperación automática de errores* permite a un analizador sintáctico realizar un diagnóstico apropiado y una modificación de la entrada errónea, de modo tal, que permita un análisis sintáctico correcto. Estas técnicas fueron introducidas el año 1963 [Iro63] y, a partir de entonces, se han desarrollado y evolucionado considerablemente [DP95].

En este capítulo presentamos algunos de los conceptos subyacentes al tratamiento automático de errores, junto con algunas de las técnicas más importantes. Describimos, además, la metodología utilizada en la implementación realizada en este trabajo, la cual está basada principalmente en la desarrollada por Burke y Fisher [BF87].

3.1. Conceptos importantes

Las distintas metodologías existentes para la recuperación de errores introducen una serie de *técnicas*, que pueden ser clasificadas de acuerdo a la estrategia que utilizan para proseguir el análisis frente a un error. Algunas de éstas simplemente

descartan parte de la entrada, mientras que otras la modifican hasta conseguir reparar los errores. Estas modificaciones pueden ser simples e involucrar tan solo la inserción, eliminación o reemplazo de un símbolo; o ser más complejas e involucrar una modificación de una porción considerable de la entrada.

A continuación, definimos algunos conceptos fundamentales y describimos los distintos tipos de técnicas existentes.

3.1.1. Definiciones

Considérese un analizador sintáctico y su configuración en un instante cualquiera. Definimos como `TOKENS` a la lista de tokens que aún no han sido analizados sintácticamente. El *token actual* es el primer elemento en `TOKENS` y representa a aquél que ha de ser analizado sintácticamente a continuación. En caso de presentarse un error sintáctico, nos referimos al *token actual* como *token de error*. Debe considerarse que, como mostramos en la sección 3.3.1, el *token de error* no necesariamente es el *token erróneo*.

3.1.2. Técnicas correctivas y no correctivas

Dentro de las técnicas de recuperación de errores se pueden distinguir dos clases: las técnicas *correctivas* y las *no correctivas* [DP95]. Una técnica no correctiva tiene como propósito principal descartar la entrada errónea y continuar con el análisis de modo tal de notificar, en una sola pasada del analizador, todos los errores que se encuentren. Por su parte, las técnicas correctivas, frente a la presencia de un error, efectúan modificaciones al texto de entrada hasta encontrar una modificación que logre corregirlo.

En este trabajo nos concentramos en las técnicas correctivas, ya que *Bison* cuenta con una técnica no correctiva basada en producciones de error [DS02], por lo que no es de nuestro interés profundizar en el estudio de este tipo de técnicas.

3.1.3. Fases del tratamiento de errores

El proceso de tratamiento de un error se puede descomponer en tres fases. La *detección del error* ocurre cuando el analizador sintáctico es incapaz de encontrar una acción posible a partir de su configuración. Entonces se efectúa un *manejo del error*,

en el cual se llevan a cabo distintas modificaciones en búsqueda de una configuración que permita al analizador sintáctico encontrar una acción factible. Típicamente, estas modificaciones se limitan a la modificación del texto de entrada que rodea al punto donde se detectó el error. Si el texto a modificar se encuentra en un punto anterior al token de error, puede ser necesario deshacer parte de las acciones de análisis sintáctico que se han llevado a cabo hasta ese momento. Cómo se maneje esta operación puede influir bastante en la eficiencia del analizador sintáctico. En la sección 3.3.1 detallamos la técnica propuesta por Burke y Fisher para manejar este problema adecuadamente. Finalmente, es necesario *reportar el error* al usuario. Este reporte debe incluir información que permita al usuario identificar el bloque de código donde se encuentra el error. En caso de haber efectuado una reparación, debe ser informada.

3.1.4. Tipos de correcciones

Una técnica correctiva modifica el texto de entrada hasta conseguir que pueda ser analizado por el analizador sintáctico. Podemos clasificar los tipos de reparaciones correctivas, dependiendo del tipo de modificación que se intente hacer para corregir el error presente. Por una parte, se encuentran las *técnicas locales*, que intentan modificar una pequeña porción del contexto alrededor del token de error. Por otro lado, se encuentran las *técnicas globales*, que utilizan una parte mayor de contexto para determinar el tipo de reparación a aplicar. Por último, se encuentran las *técnicas de reparación secundarias* que intentan corregir los errores presentes al descartar parte de la entrada.

A continuación describimos las reparaciones que presentan mayor interés en nuestro trabajo. Describimos primero algunas técnicas de reparación local, para luego describir brevemente las técnicas globales y secundarias.

Reparación simple

Esta técnica de reparación local fue introducida por Irons en el año 1963 [Iro63]. Una *reparación simple* consiste en la eliminación, inserción o modificación de un token de entrada o la mezcla de dos tokens adyacentes, de modo tal de obtener un fragmento de código que pueda ser analizado sintácticamente.

Ejemplo 3.1 (reparación simple). Considérese el fragmento de código en C pre-

Listado 3.1: Código en C con un error simple.

```
1 for (i = 0; i < n, i++) {
2     *(m + i) = 0;
3 }
```

sentado en el Listado 3.1. Este código presenta un ciclo `for` con un pequeño error sintáctico que puede ser corregido mediante una reparación simple.

De acuerdo con la gramática del lenguaje C [Deg95], se esperan dos expresiones terminadas por un *punto y coma* dentro de los argumentos de un ciclo `for`. El error se detecta al analizar el paréntesis derecho, dado que, a falta de un segundo *punto y coma*, la expresión `i < n, i++` no puede ser reducida. Una inspección del texto antes del token de error nos permite percatarnos de que el reemplazo de la coma por un *punto y coma* permite la reducción de la expresión `i < n;` y posteriormente la reducción de `i++`, con lo que el análisis puede avanzar y el error ha sido corregido.

Las primeras técnicas de reparación simple proponen sólo la utilización de símbolos terminales como potenciales reparaciones. Otros métodos más recientes adoptan también el uso de no terminales [Cha91b].

Reparación de ámbito

Uno de los errores comunes en los que se incurre durante la programación consiste en la supresión de alguna secuencia de tokens que cierre un ámbito dentro de un programa. Ejemplos de estas secuencias *cerradoras* son las llaves utilizadas en un programa escrito en C o la palabra clave `END` en Pascal. La *reparación de ámbito*, introducida por Burke y Fisher [BF87], consiste en la inserción de una secuencia de tokens que cierran un ámbito en el lugar donde se detecta el error. Esta reparación considera la inserción de múltiples tokens, debido a que algunos lenguajes pueden considerar como *cerradores* no sólo a un token, sino que a una secuencia.

Ejemplo 3.2 (reparación de ámbito). Para ejemplificar esta técnica, observemos el código Ada del Listado 3.2. La gramática de Ada [Ada95] requiere que toda expresión `if` finalice con la expresión `end if;`. Una vez que el analizador sintáctico encuentra el token `loop`, se presenta un error, dado que en realidad se espera un `if` que permita la reducción del bloque `if ... end if;`.

Listado 3.2: Código de Ada con un error de ámbito.

```
1 procedure P is
2 begin
3   loop
4     if X > 0 then
5       Y := 2;
6   end loop;
7 end p;
```

Una corrección simple para este problema no sería suficiente, ya que la inserción de solo un token no basta para corregir el error. La eliminación del token `loop` y la posterior inserción de `if` en su lugar, corregiría el error localmente, pero se produciría un nuevo error de similares características al llegar al token `p`. Sin embargo, una inserción del cerrador `end if;` un token a la izquierda del token de error, produciría una corrección óptima.

La técnica original de Burke y Fisher requiere la especificación de las secuencias cerradoras por parte del diseñador del analizador. Posteriormente, Charles propuso un método para construir el conjunto de cerradores de manera automática, a partir de la especificación de la gramática de entrada [Cha91b].

Recuperación a nivel de frase

La *recuperación a nivel de frase* es una técnica que involucra la recopilación y posterior reemplazo o eliminación del contexto donde se encuentra un error sintáctico, de modo tal de que la nueva configuración permita al analizador avanzar. Típicamente, se desea modificar el contexto más pequeño posible que permita al analizador proseguir en el texto de entrada.

El contexto erróneo es típicamente conocido como *frase de error*. Este contexto se identifica durante la primera parte de la recuperación, conocida como *fase de condensación*, durante la cual se recopila contexto tanto a la izquierda como a la derecha del token de error. En particular, la recopilación de contexto a la izquierda es costosa y, si bien era popular dentro de las primeras técnicas de recuperación a nivel de frase [GR75], técnicas más recientes priorizan la recopilación de contexto a la derecha, dado que este proceso no requiere deshacer el análisis sintáctico [SSS83, DP95].

Cada intento de recuperación a nivel de frase consiste en tomar una porción del contexto alrededor del token de error y reemplazarla por un símbolo o simplemente descartarla. Si la configuración resultante permite que se efectúe una acción de análisis de manera correcta, se considera al contexto descartado o reemplazado como *frase de error*.

Debe notarse que la recuperación simple, como la hemos presentado previamente, es un caso particular de recuperación a nivel de frase. Si la frase de error es vacía y se inserta un no terminal, la recuperación simple equivalente corresponde simplemente a la inserción de un no terminal. Por otra parte, si la frase de error consiste en sólo un terminal y éste es descartado, entonces esta acción equivale a una recuperación simple mediante eliminación de un token de entrada. Finalmente, si se elimina un terminal y se inserta un terminal distinto, entonces estamos en presencia de una recuperación simple mediante reemplazo de un token de entrada.

Recuperación Global

Las técnicas de *recuperación global* se distinguen de las de recuperación local en que utilizan un fragmento extenso del contexto para determinar la mejor reparación posible. Sin embargo, pese a que los resultados que producen son mejores que los obtenidos mediante reparación simple, el costo de cómputo que inducen es considerablemente mayor [DP95].

Recuperación Secundaria

La *recuperación secundaria* es una técnica que consiste en desechar una porción del texto de entrada, tanto antes como después del token de error. En general, toda técnica de reparación secundaria debe ser capaz de asegurar que el análisis pueda proseguir correctamente.

Un ejemplo particular de reparación secundaria es la conocida como recuperación en *modo de pánico* [ALSU06]. En este tipo de reparación, se debe definir un conjunto de *tokens de sincronización* apropiados, tal como un *punto y coma* en C o un `end if` en Ada. El método procede descartando todo el texto de entrada desde el punto del error hasta el siguiente token de sincronización y luego eliminando todos los elementos en la pila del análisis hasta encontrar uno que, en conjunto con el token de sincronización, permita que el análisis continúe. De esta manera, el analizador

simula que ha podido analizar correctamente el bloque de texto que ha eliminado.

Dada la naturaleza no correctiva de la recuperación secundaria, su uso se da en varios métodos una vez que las técnicas correctivas han fallado [BF87].

3.2. Metodologías para la corrección de errores

A partir de un conjunto de técnicas cuidadosamente seleccionadas, puede definirse una *metodología para la corrección de errores*. Dependiendo de las técnicas que la compongan, una metodología permitirá enfrentar el problema de la corrección de errores sintáctico con distintos niveles de satisfacción.

Las diversas metodologías pueden ser analizadas de acuerdo a las siguientes características [DP95]:

Eficiencia: El tiempo de ejecución de una metodología, al igual que la memoria que requiera, determinan su *eficiencia*. Idóneamente, una metodología no debe incrementar considerablemente las necesidades de recursos del analizador sintáctico que la implementa. Además, estos requerimientos deben ser independientes del tamaño de la entrada a procesar.

Independencia del lenguaje: La capacidad de ser igualmente eficaz para distintos tipos de lenguajes se conoce como *independencia del lenguaje*. Esta característica es vital en la implementación de una técnica de corrección de errores en un generador de analizadores sintácticos.

Flexibilidad: La capacidad de una metodología de permitir un ajuste de parámetros o la especificación de información adicional que mejore el desempeño de la rutina de corrección de errores es conocida como *flexibilidad*.

En base a estas características, presentamos las metodologías más importantes que se han desarrollado y las comparamos.

3.2.1. Irons

Irons propuso en el año 1963 [Iro63] una técnica de análisis sintáctico que incorpora una metodología para la corrección simple de errores. La técnica de análisis

sintáctico es no-determinista, es decir, se mantienen simultáneamente todas las alternativas de análisis y solo se descartan en presencia de un símbolo que no permita al analizador continuar.

La corrección se efectúa cuando, para un cierto token de entrada, se descartan todas las secuencias de análisis que se habían mantenido hasta ese paso. Entonces, el método procede mediante la inserción, eliminación o sustitución local de tokens, hasta obtener una configuración que repare el error.

El método tiene la característica de que, independientemente de la cantidad de errores presentes en el texto de entrada, éste será manipulado tanto como sea necesario, hasta obtener un fragmento de código sintácticamente correcto.

3.2.2. Pennello y DeRemer

Pennello y DeRemer propusieron en 1978 una técnica de reparación de errores a nivel de frase compuesta por dos etapas: *avance* y *reparación* [PD78].

En la etapa de avance se recopila contexto a la derecha del token de error por medio de un *Autómata de Avance (Forward Move Automaton)*, hasta que se encuentre un nuevo error sintáctico o se intente reducir por sobre el símbolo erróneo. El autómata recopila todas las alternativas de análisis posibles con respecto a los símbolos en la entrada que aún no han sido procesados.

En la fase de reparación, se intenta asociar cada una de las secuencias de configuraciones obtenidas por el FMA con el contexto donde se encuentra el error sintáctico, levemente modificado. Si alguna de las secuencias de configuraciones obtenidas corresponde con el contexto erróneo, levemente modificado, entonces se acepta la modificación realizada como reparación. Estas reparaciones pueden consistir en la eliminación, inserción o modificación de un token.

3.2.3. Burke y Fisher

La metodología de Burke y Fisher, propuesta en 1987 [BF87], combina técnicas de reparación simple, de ámbito y secundaria y ha probado ser una de las más eficientes [CPRT02, DP95].

En una primera fase, la técnica realiza intentos de reparación simple en el contexto a la izquierda del token de error, eliminando, insertando tokens, o mezclando tokens adyacentes. Para evitar la necesidad de deshacer acciones semánticas, Burke y Fisher

propusieron el uso de *postergación del análisis sintáctico*, una técnica que consiste en mantener una pila temporal de análisis en el contexto izquierdo del punto que se está analizando, de modo tal que para los elementos en esa pila se ha realizado análisis pero no se han ejecutado acciones semánticas. Esta pila se mantiene para los k símbolos a la izquierda del punto de análisis y, cada vez que avanzamos, se elimina el k-ésimo símbolo de la pila y se ejecutan las acciones asociadas a ella.

Si la reparación simple falla, el método prosigue con reparación de ámbito. En esta fase se intenta insertar secuencias *cerradoras* en las mismas posiciones donde se intenta realizar reparaciones simples. Si la inserción de una secuencia cerradora corrige el error y permite al analizador avanzar, pero no lo suficiente como para considerar la reparación satisfactoria (i.e., se detecta un nuevo error), entonces se invoca a este método de manera recursiva. Dado que las secuencias cerradoras dependen del lenguaje a analizar, el método de Burke y Fisher requiere que estas secuencias sean especificadas como parámetros del analizador.

Si la reparación de ámbito también es infructuosa, entonces se procede con un mecanismo de recuperación secundaria, en la cual se descarta una secuencia de tokens, de modo tal que el análisis pueda proseguir.

En la sección 3.3 discutimos en mayor profundidad los detalles particulares de esta técnica.

3.2.4. Charles

La técnica propuesta por Charles [Cha91a] está basada en la de Burke y Fisher (sección 3.2.3). Está compuesta por tres fases: recuperación simple, de ámbito y secundaria. Sin embargo, Charles propone mejoras que hacen del método más eficiente, tanto en tiempo y espacio y, además, lo automatizan completamente.

En este método, durante la etapa de recuperación simple, no sólo se considera la inserción y eliminación de tokens, sino que además la de no terminales. Además, se propone el uso de un controlador de análisis LR conocido como *controlador con posposición*, que mejora el rendimiento del analizador con respecto a la técnica de Burke y Fisher.

Otra ventaja del método propuesto por Charles, en comparación con la metodología de Burke-Fisher, se relaciona con la reparación de ámbito. En el método de de Burke y Fisher es necesario especificar explícitamente el conjunto de secuencias

cerradoras para cada lenguaje. Charles, por su parte, propone un método para la generación automática del conjunto de cerradores a partir de la gramática de entrada. Además, propone el uso de *coincidencia de patrones* para determinar el cerrador apropiado para corregir un determinado error, en vez de insertar cada uno de los cerradores hasta obtener un análisis sintáctico satisfactorio.

3.3. Técnica de Burke-Fisher

Para el curso de este trabajo hemos decidido implementar en *Bison* la técnica de Burke-Fisher por las siguientes razones. Primero, la literatura reciente muestra esta técnica como una de las más efectivas existentes [CPRT02, DP95]. Segundo, el generador de analizadores sintácticos *ML-Yacc* para el lenguaje *Standard ML* implementa también esta técnica en su infraestructura para la recuperación automática de errores [TA00]. Finalmente, existe consenso entre los desarrolladores actuales de *Bison* en la conveniencia del uso de esta metodología para brindar un tratamiento de errores de alta calidad a sus usuarios. En particular, la implementación de la técnica de Burke-Fisher en *Bison* fue propuesta por los mantenedores de *Bison* como uno de los posibles proyectos para el programa *Summer of Code* de *Google*, durante las ediciones 2006[1] y 2007[2].

En la sección 3.2.3 hemos descrito brevemente la técnica de Burke-Fisher. A continuación, describimos en más detalle la metodología de *postergación de las acciones del análisis sintáctico* y el funcionamiento de la metodología para cada una de los tipos de reparación que implementa.

3.3.1. Postergación de las acciones del análisis sintáctico

En la práctica, el token de error no siempre es exactamente el token erróneo. Esto puede significar que, para algunos errores sintácticos, sea necesario deshacer parte de las acciones del análisis para poder corregir los errores adecuadamente.

Ejemplo 3.3 (error sintáctico que requiere deshacer acciones del análisis). Considérese el Listado 3.3, que presenta un error que es detectado tardíamente por el analizador.

[1] `http://www.gnu.org/software/soc-projects/ideas-2006.html#bison`
[2] `http://www.gnu.org/software/soc-projects/ideas-2007.html#bison`

Listado 3.3: Error simple en un fragmento de código de Pascal.

```
1  PROGRAM P;
2   VAR X: INTEGER,
3  BEGIN
4    X := 20;
5    IF X = 1 THEN
6      IF X = 1 THEN
7        x := 1
8      ELSE
9        WRITELN('␣');
10   ELSE
11     x := 2
12 END
```

El *token de error* para este fragmento de código es la palabra clave `ELSE`, que sigue al *punto y coma*. La eliminación de esta palabra clave produciría un programa sintácticamente correcto, sin embargo, la semántica del programa cambiaría fuertemente. Otra corrección posible y menos invasiva, es la eliminación del *punto y coma* en la línea 9. Este fragmento muestra un error típico cometido por programadores de C que son nuevos en Pascal: el uso de un *punto y coma* en Pascal está reservado para separar expresiones y no para terminarlas.

Analicemos el estado de la pila previo al desplazamiento del *punto y coma*. Ésta contendrá los estados correspondientes al prefijo viable `[ stmt_list ‘;’’ IF expression THEN statement ELSE statement ]`. Para una gramática tradicional de Pascal, el desplazamiento del `punto y coma` producirá una reducción a `[ stmt_list ‘;’’ stmt ]` y finalmente a `[stmt_list]`. De existir acciones semánticas, a esta altura el analizador habrá ejecutado aquéllas asociadas con los no terminales `stmt_list` y `stmt`. Al intentar analizar el `ELSE` de la línea siguiente, el analizador determina que `[ stmt_list ELSE ]` no es un prefijo viable y detecta el error sintáctico.

Para intentar reparar este error mediante la eliminación del *punto y coma* a la izquierda del token de error, necesitamos deshacer las reducciones que lo absorbieron. Desafortunadamente, deshacer estas reducciones es costoso y, peor aún, deshacer las acciones semánticas realizadas durante las reducciones puede requerir de un procedimiento específico y, en algunos casos, puede llegar a ser imposible.

El método de Burke y Fisher se encarga de simular la capacidad de deshacer

acciones de análisis sintáctico mediante una técnica llamada *postergación de las acciones semánticas*. Esta técnica permite simular la capacidad de deshacer k acciones al mantener una cola con los $k-1$ últimos tokens analizados, los cuales se conocen como *tokens postergados*. Para cada uno de estos símbolos no se han ejecutado las acciones semánticas aún. Cada vez que se desplaza el k-ésimo símbolo, se ejecutan las acciones del análisis sintáctico correspondientes al primer elemento de la cola de tokens postergados y sus correspondientes acciones semánticas.

Esto puede verse como *doble análisis sintáctico*: uno de los analizadores verifica la correctitud del texto de entrada sin ejecutar acciones semánticas. El otro analizador se mantiene $k-1$ tokens por detrás, tiene siempre texto sintácticamente correcto y, además, ejecuta las acciones semánticas.

3.3.2. Reparación Simple

Como ya mencionamos brevemente en la sección 3.2.3, la primera etapa de la técnica de Burke y Fisher consiste en reparación simple. Esta reparación contempla la inserción, eliminación y mezcla de tokens adyacentes en una vecindad del token de error, de modo tal de generar *reparaciones candidatas*, que luego son evaluadas para determinar la mejor corrección posible.

Estado de un analizador

El estado de un analizador se puede describir por dos componentes: una pila de estados y la secuencia de tokens que no ha sido procesada. Adicionalmente, el método requiere de una pila de análisis sintáctico, compuesta por los símbolos gramaticales que ya han sido reconocidos. Esta pila puede ser construida directamente, a partir de la pila de estados, dado que a cada estado del analizador le corresponde un símbolo gramatical. Si, además, consideramos un grado de postergación de las acciones del análisis sintáctico de $k+1$ tokens, el analizador debe mantener una secuencia de k tokens postergados, para los cuales no se han efectuado acciones semánticas. Al detectar un error sintáctico en el token actual, las reducciones que preceden el desplazamiento del primer token postergado no han sido aplicadas a la pila de estados, por lo que ésta puede ser regenerada para que corresponda a la configuración de la pila del análisis, simulando, entonces, que el análisis se deshace.

Definamos el contexto izquierdo del analizador en el momento de encontrar un

error. Este contexto se compone por una concatenación de los símbolos en la pila de análisis y los símbolos pospuestos,

$$l_1 \dots l_t l_{t+1} \dots l_{t+k}$$

donde $l_i = p_i$, con $1 \leq i \leq t$ es el i-ésimo elemento de la pila del análisis, y $l_{t+j} = d_j$, con $1 \leq j \leq k$ es el j-ésimo token postergado. En lo que sigue, denotamos a esta tupla como `LEFT_CONTEXT`.

Se define una *prueba de reparación simple* como la prueba de todas las alternativas de reparación simple posibles en un punto l_T de `LEFT_CONTEXT`. En general, se lleva a cabo una prueba en el token de error y en una secuencia de uno o más de los tokens postergados que le preceden. Para una progresión eficiente entre evaluaciones, Burke y Fisher sugieren que esto se realice partiendo por el elemento más a la izquierda donde se quiera efectuar una prueba y terminar con el token de error, que es el elemento más a la derecha.

Sea l_i el punto donde se efectuará la siguiente prueba de reparación simple. Entonces debemos preparar el entorno como sigue: si l_i es un elemento de la pila de análisis, entonces $p_{i+1} \dots p_t$ se eliminan de la pila de análisis; si l_i es uno de los tokens postergados, entonces las acciones asociadas con $d_1 \dots d_{i-1}$ se aplican a la pila de análisis. En cualquiera de ambos casos, la pila de estados debe reconstruirse para ser equivalente al estado de la pila de análisis.

A continuación, es necesario tomar el sufijo de `LEFT_CONTEXT` que comienza con l_i y concatenarlo como prefijo de `TOKENS`. Esto define el estado del analizador sintáctico antes de comenzar con la primera prueba.

Prueba de recuperación simple

Sea l_i el token donde realizamos a continuación una prueba de recuperación simple. De acuerdo con lo señalado en la sección anterior, la configuración actual de `TOKENS` es $l_i \dots e \dots l_n$, donde n es la cantidad de tokens por analizar y e es el token de error. Durante esta evaluación, se genera un conjunto de reparaciones candidatas incluyendo inserción de tokens antes de l_i, la substitución de l_i por otro token o la eliminación de l_i. Sea t un terminal considerado candidato para inserción. Entonces, t se concatena frente a `TOKENS`, resultando $tl_i \dots l_n$. Si t es candidato para sustitución, entonces `TOKENS` resulta en $tl_{i+1} \dots l_n$. Si la eliminación es una evaluación candidata,

entonces `TOKENS` se transforma en $l_{i+1} \dots l_n$. Otra reparación candidata es la mezcla de dos tokens adyacentes.

Una vez que `TOKENS` ha sido modificado, de acuerdo a la evaluación a realizar, se efectúa un *análisis sintáctico de prueba* para determinar la cantidad de tokens que pueden consumirse antes de llegar a un nuevo error. Si el análisis logra avanzar al menos $t_{\min}$ tokens, entonces la reparación evaluada se añade a un conjunto de *reparaciones candidatas*, que llamamos `CANDIDATES`.

Evaluación de las reparaciones candidatas

Al finalizar cada una de las evaluaciones que generan las reparaciones candidatas, `CANDIDATES` se evalúa para determinar si existe una reparación simple que pueda llevarse a cabo. Esta evaluación consiste en la selección de una de las reparaciones candidatas o la eliminación de todas ellas. Se puede realizar una poda de las reparaciones candidatas, en base a la cantidad de tokens que cada candidata permite analizar por sobre el token de error. En particular, pueden eliminarse todas las reparaciones candidatas que no logren analizar $t_e \geq t_{\min}$ tokens por sobre el token de error y, en caso de que ninguna lo logre, se consideran sólo aquellas que logran analizar la mayor cantidad de tokens posible. Si finalmente hay más de una alternativa presente, se elige una de ellas siguiendo un orden de preferencia heurístico determinado: mezcla, corrección ortográfica, inserción, eliminación, sustitución. Este orden ha determinado los mejores resultados, pero no necesariamente ha de ser el orden elegido en una implementación [BF87].

Si finalmente no se consigue una reparación candidata que cumpla con las condiciones precisadas, entonces no se realiza una reparación simple y es necesario avanzar a la reparación de ámbito.

3.3.3. Reparación de ámbito

La reparación de ámbito en la metodología de Burke y Fisher funciona de un modo bastante similar al modo de reparación simple que contempla inserción de símbolos, salvo que típicamente involucra la inserción de secuencias de uno o más tokens necesarios para cerrar un ámbito sintáctico abierto (sección 3.1.4).

Una *evaluación para reparación de ámbito* se efectúa en cada punto donde una evaluación para reparación simple se lleva a cabo (sección 3.3.2) y la progresión entre

evaluaciones se lleva a cabo de la misma manera que en la reparación simple.

Cada secuencia cerradora se inserta en el punto anterior al token actual y se evalúa si el análisis sintáctico puede avanzar más allá de la secuencia insertada. De no ser así, la secuencia se rechaza. Si el análisis logra avanzar por sobre la secuencia insertada y un token más por delante del token de error, entonces la secuencia evaluada se acepta como la recuperación. Si el análisis logra avanzar por sobre la secuencia insertada, pero no lo suficiente como para aceptarla como recuperación, se invoca recursivamente el procedimiento de recuperación de ámbito para cerrar eventuales secuencias múltiples de cerradores que estén todavía abiertas. Todas las secuencias candidatas se evalúan de esta manera y, eventualmente, o se ha aceptado una como reparación o todas han fallado y es necesario invocar el procedimiento para recuperación secundaria.

3.3.4. Reparación secundaria

La recuperación secundaria en el método de Burke y Fisher consiste en continuar el análisis sintáctico al descartar una secuencia de tokens que precedan inmediatamente el token de error o una secuencia que comience con él.

De acuerdo con el análisis de Burke y Fisher, en esta etapa la mejor calidad de reparaciones se puede obtener mediante postergación de las acciones semánticas de un sólo token. Por lo tanto, al preparar la configuración del analizador para la recuperación secundaria, es necesario vaciar la cola de tokens postergados, ejecutando así las acciones semánticas correspondientes.

Comenzando con el token de error, se verifica si es posible realizar el análisis sintáctico al descartar parte de la pila de estados, además de insertar una o más secuencias de tokens cerradores. Para esto, se efectúa un descenso por la pila de estados, verificando en cada punto, si es posible continuar el análisis sintáctico considerando sólo los estados hasta el actual. Si esto es posible, entonces se descartan los estados por sobre el actual en la pila de estados. En caso contrario, se desciende un estado más en la pila. Esta iteración se lleva a cabo en toda la pila de estados.

Dependiendo de la implementación, se debe verificar si el análisis puede avanzar una cantidad mínima de tokens más allá del token de error. Si para una iteración no es posible continuar el análisis, entonces el token actual es eliminado de `TOKENS`, lo que transforma al token siguiente en el token actual. La iteración sobre la pila

de estados entonces es reanudada. Finalmente, habrá un punto en el cual o una reparación secundaria es satisfactoria o se habrán eliminado todos los tokens restantes por analizar.

4. Metodología de trabajo

Bison es un Software Libre, lo cual significa, en parte, que el código fuente puede ser accedido y modificado por cualquier parte interesada. Esto hace posible que nuestro trabajo se realice de manera completamente independiente y que, eventualmente, nuestro trabajo pueda ser adaptado por el proyecto. Este código fuente es mantenido en el repositorio CVS del proyecto GNU. El desarrollo de este proyecto se coordina mediante listas de correo públicas, a través de las cuales, tanto desarrolladores como usuarios, se comunican para decidir el curso del desarrollo.

Uno de nuestros objetivos principales a mediano plazo es lograr que nuestra implementación sea adoptada por el proyecto *Bison*. En la sección 4.1 explicamos la metodología a usar para hacer esta eventual adopción lo más simple posible, además de las distintas fases en las que se descompone el desarrollo. En la sección 4.2 describimos brevemente las distintas herramientas utilizadas durante el curso de nuestro trabajo.

4.1. Metodología de desarrollo

La implementación de una característica de software en paralelo a su desarrollo, implica una decisión importante acerca de la versión del código sobre la que se trabaja. Cuando uno de los objetivos finales del trabajo es la integración de esta característica, implementada en una rama paralela, dentro de la rama principal de desarrollo del proyecto, lo ideal es mantener la menor cantidad de discrepancias entre estas ramas, integrando cambios a la rama principal de manera progresiva, cuando éstos ocurren. Esta integración constante puede requerir un cierto esfuerzo adicional, el cual puede ser minimizado si se utiliza una buena herramienta para el control de versiones de código.

Una alternativa que reduce el esfuerzo por integración continua a cero, consiste en trabajar sobre una versión estable del software, sin preocuparse sobre los cambios que puedan ocurrir en versiones posteriores. Si bien esto permite dedicar un 100 % de los esfuerzos al desarrollo de las características deseadas, dificulta enormemente la integración con la rama principal de desarrollo, una vez que la implementación es lo suficientemente madura.

Considerando que es de nuestro interés que nuestro trabajo, una vez completo, sea integrado con *Bison* con la mayor facilidad posible, decidimos trabajar sobre una rama paralela a la rama de desarrollo principal e integrar los cambios en ésta a nuestra rama de manera continua.

En la siguiente sección nos referimos a las herramientas utilizadas para integrar los cambios fácilmente y mantener un control de los cambios.

4.1.1. Control de cambios en el código

Para facilitar la integración continua, además de llevar un control de los cambios, utilizamos el sistema de control de versiones *Git* (sección 4.2.2).

Gracias a *Git*, contamos con una rama llamada *origin*, que se sincroniza continuamente con la rama principal en el repositorio CVS de *Bison*. Este repositorio es clonado por medio de *Git* para crear nuestro repositorio de trabajo, en el cual, además de contar con *origin*, tenemos una rama *master*, en donde realizamos todas las modificaciones.

Los cambios realizados en el repositorio CVS por los desarrolladores de *Bison*, son importados en *origin* y luego integrados con nuestras modificaciones en *master*, mediante las herramientas que *Git* provee para realizar esta tarea de manera automática.

Una de las ventajas adicionales del uso de *Git*, es que se puede utilizar un clon del repositorio de trabajo como respaldo, manteniéndolo en una ubicación distinta. Por esto, mantenemos un clon en un servidor separado, al cual se tiene acceso mediante *SecureShell*, un protocolo soportado por *Git*, haciendo el respaldo continuo una tarea trivial.

4.1.2. Etapas en el desarrollo de la solución

La metodología de Burke-Fisher requiere de una serie de características, que una vez identificadas y clasificadas, pueden ser utilizadas como hitos del desarrollo. Por esto, dividimos el proceso de desarrollo en las siguientes etapas.

Analizador con posposición

El primer hito consiste en la modificación del algoritmo de análisis y estructuras de datos relacionadas, de modo tal de permitir que el análisis sintáctico se efectúe de manera pospuesta (sección 3.3.1). Este hito se considera completo cuando se cuenta con un analizador capaz de realizar análisis sintáctico sobre cualquier entrada correcta, al mismo tiempo que mantiene almacenadas tanto una secuencia de k tokens postergados, como las secuencias de reducciones entre ellos.

Determinación de las reparaciones candidatas para la recuperación simple

Este hito consiste en la implementación del algoritmo que, mediante el uso de los tokens postergados y el estado del analizador postergado, ejecuta pruebas simples hasta determinar todas las posibles modificaciones que corregirían el error sintáctico, de existir alguna. Este hito está completo cuando se cuenta con un algoritmo capaz de proveer a la siguiente fase de un conjunto de correcciones candidatas.

Extensión del formato de entrada de Bison para mejorar la calidad de las recuperaciones

Tanto el método de recuperación simple como el de recuperación de ámbito requieren de información adicional, provista por el diseñador del analizador, para mejorar la calidad de sus recuperaciones. Este hito consiste en extender el formato de entrada de *Bison* para permitir al usuario proveer, mediante directivas, de esta información en el archivo que especifica la gramática. Esta información luego es entregada a los analizadores sintácticos generados, para permitir que puedan alimentarse de ella y mejorar la calidad de la recuperación.

Búsqueda de la mejor reparación candidata para la recuperación simple

Una vez que se cuenta con una serie de reparaciones candidatas, es necesario evaluarlas y determinar la mejor reparación posible. Este hito consiste en la implementación de la heurística que evalúa las reparaciones candidatas y decide cuál de ellas ha de ser elegida. Una vez completo, se cuenta con la mejor corrección simple posible o se ha logrado determinar que ninguna corrección simple cumple con los requisitos mínimos para ser aceptada y es necesario recurrir a la reparación de ámbito.

Aplicación de la recuperación simple elegida y reanudación del análisis sintáctico

Una vez elegida la recuperación, es necesario aplicarla y permitir la reanudación del análisis sintáctico, devolviendo el control al algoritmo principal de análisis. Una vez completado este hito, se cuenta con un analizador sintáctico capaz de recuperarse de errores simples.

Elección de una reparación de ámbito

Esta etapa consiste en la implementación necesaria para, en caso de que la etapa de reparación simple falle, probar las secuencias cerradoras del lenguaje y determinar cuál de ellas permitiría corregir el error sintácticamente.

Aplicación de la recuperación de ámbito elegida y reanudación del análisis

Este hito consiste en la implementación requerida para, una vez determinada la corrección de ámbito a realizar, aplicarla y devolver el control al analizador sintáctico. Una vez completado este hito, se cuenta con un analizador capaz de corregir los errores sintácticos, ya sea mediante reparación simple o reparación de ámbito.

4.1.3. Enfoque para una implementación efectiva

Considerando que nos encontramos frente al problema de modificar un software para la generación de analizadores sintácticos, la tarea puede volverse en extremo compleja, ya que además de la implementación de las técnicas y algoritmos requeridos, es necesario comprender los procesos de generación de código involucrados y

el procesamiento de los archivos de entrada. Es por esto que hemos decidido utilizar un enfoque desde lo general hacia lo particular, que permita ir profundizando en el nivel de detalle, a medida que se maduran los conocimientos sobre el funcionamiento interno tanto de *Bison*, como de los analizadores sintácticos que genera.

En concreto, el trabajo de implementación comienza con el estudio del funcionamiento y posterior modificación de un analizador sintáctico generado por *Bison* para un lenguaje sencillo. De esta manera, comenzamos a resolver el problema para un caso particular, que nos permite interiorizarnos en los detalles propios de los analizadores generados por *Bison*. En esta etapa se modifica el algoritmo de análisis, para obtener un algoritmo de análisis con postergación de las acciones.

Una vez completo el desarrollo del analizador con posposición, es posible integrar los cambios realizados en la plantilla LALR de *Bison* (introducción al capítulo 5), de modo tal, que cualquier analizador sintáctico generado cuente con la característica de posposición.

La implementación de las rutinas de recuperación puede hacerse directamente en la plantilla, una vez que ya se comprende en profundidad cómo funciona el algoritmo de análisis sintáctico.

De la misma manera, algunas características de los analizadores sintácticos que no son fundamentales para su funcionamiento, como las acciones semánticas y las ubicaciones (sección 2.3) pueden ser obviadas en la fase inicial y ser añadidas paulatinamente, a medida que la implementación madura.

En particular, el código referente a las ubicaciones solo se hace presente en el analizador sintáctico generado si el usuario las requiere explícitamente. Por esto, es posible iniciar la implementación sin considerar las ubicaciones y agregarlas en una etapa posterior del desarrollo.

4.2. Herramientas

A continuación describimos brevemente las herramientas utilizadas durante la implementación. Estas herramientas consisten básicamente en herramientas para desarrollo de software tradicionales en ambientes Unix, y los lenguajes C y M4.

Todas las herramientas utilizadas son Software Libre y pueden ser descargadas desde Internet y utilizadas sin costos ni restricciones.

4.2.1. Lenguajes utilizados

Bison está programado en el lenguaje C y para la generación de los analizadores sintácticos utiliza el procesador de macros M4. A continuación describimos brevemente ambos lenguajes.

C

C es un lenguaje de programación desarrollado por Dennis Ritchie en los laboratorios Bell durante los años '70. Es un lenguaje altamente versátil y muy utilizado para el desarrollo de aplicaciones Unix/Linux [Rit93].

Bison está escrito y además genera analizadores sintácticos LALR en el lenguaje C. La plantilla donde se implementa la recuperación automática de errores sintácticos contiene en su mayor parte código en este lenguaje.

M4

M4 es un procesador de macros que, básicamente, copia su entrada hacia la salida, expandiendo macros a medida que aparecen en la entrada. Las macros pueden ser definidas tanto por el usuario o por parte del procesador, lo cual facilita bastante la generación de código [Fre07].

Este procesador de macros es usado por *Bison* para la generación de los analizadores sintácticos que produce. Básicamente, *Bison* cuenta con un conjunto de plantillas que combinan código estático en C y macros M4, que son expandidas durante la generación de código en base a las características particulares del analizador en cuestión.

4.2.2. Otras herramientas

En adición a los lenguajes ya descritos, utilizamos dos herramientas importantes para asistir el desarrollo. El sistema de control de versiones *Git* y el generador de escáneres léxicos *Flex*.

Git

Originalmente desarrollado para llevar el control de versiones de *Linux*, *Git* es un sistema distribuido de control de versiones, que permite una administración descen-

tralizada de los cambios en el código fuente [T+08].

Flex

Flex es una herramienta similar a *Bison*, utilizada para la etapa de análisis léxico. A partir de la especificación de un conjunto de expresiones regulares, *Flex* genera un módulo en C capaz de reconocer cada uno de los distintos tokens en su entrada. La herramienta encargada del análisis sintáctico llamará a la función provista por el código generado por *Flex*, cada vez que requiera un nuevo token [PEM07].

En nuestro trabajo, hemos utilizado *Flex* para generar los analizadores léxicos que alimentan a los analizadores sintácticos generados para la depuración de nuestra implementación. De la misma manera, *Flex* nos permite facilitar la creación de pruebas para la metodología implementada.

5. Implementación

Bison es capaz de generar analizadores sintácticos de diverso tipo. Algunos de los lenguajes de destino soportados son C, C++ y Java. Además, para cada uno de esos lenguajes pueden generarse distintas clases de analizadores, incluyendo LALR y GLR.

Esto es posible, dado que *Bison* está diseñado de modo tal que se separan las distintas fases de la generación de analizadores sintácticos en un *frontend*, *núcleo* y *backend*. El *frontend* se encarga del procesamiento de los archivos de entrada y de generar una representación interna para *Bison* de la gramática y sus características. Esta representación interna es procesada por el *núcleo* de *Bison* para producir las tablas necesarias para el análisis sintáctico, de acuerdo con el tipo de analizador seleccionado por el usuario. Finalmente, el *backend* se encarga de generar el código del analizador en un archivo de código fuente, además de otros posibles formatos.

Para facilitar la generación del archivo de salida, *Bison* se apoya en *plantillas de analizadores sintácticos*. Estas plantillas son archivos que mezclan código fuente en el lenguaje de destino y macros M4 y contienen la implementación de los algoritmos necesarios para el análisis sintáctico de la clase y en el lenguaje de destino. Existen, entre otras, plantillas LALR para C, C++ y Java.

Durante la generación del código por parte del *backend*, *Bison* toma la plantilla correspondiente y se encarga de expandir las macros M4 existentes en ella. Estas macros generalmente definen algunas propiedades como las tablas de análisis, las acciones semánticas, los tipos de datos para los valores semánticos y las ubicaciones.

Este capítulo está dedicado a los detalles de la implementación de la técnica de recuperación de errores sobre la plantilla LALR para el lenguaje C. Comenzamos describiendo la estructura de los analizadores LALR que *Bison* genera, para luego

referirnos a los cambios estructurales necesarios para que éstos puedan contar con la capacidad de posponer las acciones del análisis. Luego describimos los detalles de la rutina de recuperación de errores implementada, junto con las extensiones al formato de entrada de *Bison*, que permiten a los usuarios ingresar información extendida sobre sus gramáticas para mejorar la calidad de las recuperaciones.

5.1. Estructura de los analizadores LALR generados por Bison

Antes de describir el funcionamiento de un analizador sintáctico con posposición de las acciones, es necesario comprender algunos aspectos del algoritmo de análisis sintáctico LALR de *Bison*. En esta sección, describimos brevemente las estructuras de datos involucradas y el algoritmo de análisis sintáctico LALR en *Bison*.

5.1.1. Estructuras de datos involucradas

Los tipos de datos fundamentales utilizados por los analizadores generados son definidos por *Bison* en forma genérica para asegurar, por un lado, portabilidad y, además, permitir al usuario redefinirlos en tiempo de compilación, en caso de que el tamaño de los tipos no sea suficiente. Por ejemplo, el tipo de dato utilizado para la pila de estados es `yytype_int16`, el cual es definido por defecto como un entero de 16 bits. En caso de que, para cierta aplicación, este tamaño sea demasiado pequeño para la cantidad de estados en el analizador, el usuario puede redefinir este tipo en tiempo de compilación y utilizar un tipo entero de 32 ó 64 bits.

Un analizador LALR generado por *Bison* cuenta con tres pilas: la pila de estados, la pila de valores semánticos y, opcionalmente, la pila de ubicaciones. Estas pilas están implementadas por medio de arreglos de tamaño fijo y punteros, siendo éstos reubicables en tiempo de ejecución, en caso de que el tamaño de la entrada a procesar requiera almacenar una cantidad de estados mayor al tamaño original de las pilas.

Como ya lo mencionamos previamente, la pila de estados, por defecto, es de tipo entero de 16 bits. La pila de valores semánticos es de tipo `YYSTYPE`, un tipo que por defecto es definido como entero de 32 bits, pero puede ser redefinido por el usuario como una estructura más compleja, dependiendo de su aplicación. Finalmente, la pila de ubicaciones es de tipo `YYLTYPE`, el cual está definido por defecto como una

estructura de cuatro enteros, que representan las fila y columna, tanto iniciales como finales, del símbolo gramatical correspondiente. Esta estructura también puede ser redefinida por el usuario, si así se requiere.

5.1.2. Algoritmo de control

El algoritmo implementado por *Bison* para el control del ciclo de análisis sintáctico LALR(1) es distinto del algoritmo propuesto teóricamente (sección 2.2.1). En teoría, el algoritmo LALR(1) decide el curso de acción siempre en base al contenido de la tabla *ACTION*, al token actual y al símbolo en la parte superior de la pila de análisis. Sin embargo, *Bison* utiliza una versión optimizada de este algoritmo, el cual toma la forma de una *máquina de estados finitos* (ver Figura 5.1).

Dado que típicamente las tablas *GOTO* y *ACTION* en un algoritmo LALR pueden alcanzar tamaños considerables y poco prácticos, *Bison* utiliza dos técnicas para reducir su tamaño: el uso de acciones por defecto y la compactación de las tablas.

Las *acciones por defecto* se introducen cuando, para distintos tokens, existe una acción común a partir de un mismo estado. Una acción por defecto puede ser tanto una reducción por defecto o la invocación de la rutina de error. Una *reducción por defecto* es una reducción que depende solo del símbolo en la parte superior de la pila y se efectúa independientemente del token actual. El uso de acciones por defecto reduce las dimensiones de las tablas de análisis considerablemente.

La *compactación de tablas* es una técnica que aprovecha el hecho de que las tablas de análisis son dispersas, al representarlas mediante distintas tablas de tamaño considerablemente menor, sobre las cuales se realizan distintas operaciones de bajo costo para determinar la acción a seguir [Pop06b].

En cada iteración, el algoritmo de control comienza decidiendo si es posible realizar una acción por defecto antes de leer el siguiente token desde la entrada. Si esto no es posible, entonces lee el siguiente token y, a partir de éste y el símbolo en la parte superior de la pila, determina si existe una acción por defecto a realizar. En caso de no existir una, entonces decide la acción a realizar entre una reducción, el desplazamiento del token actual y la invocación del algoritmo de error.

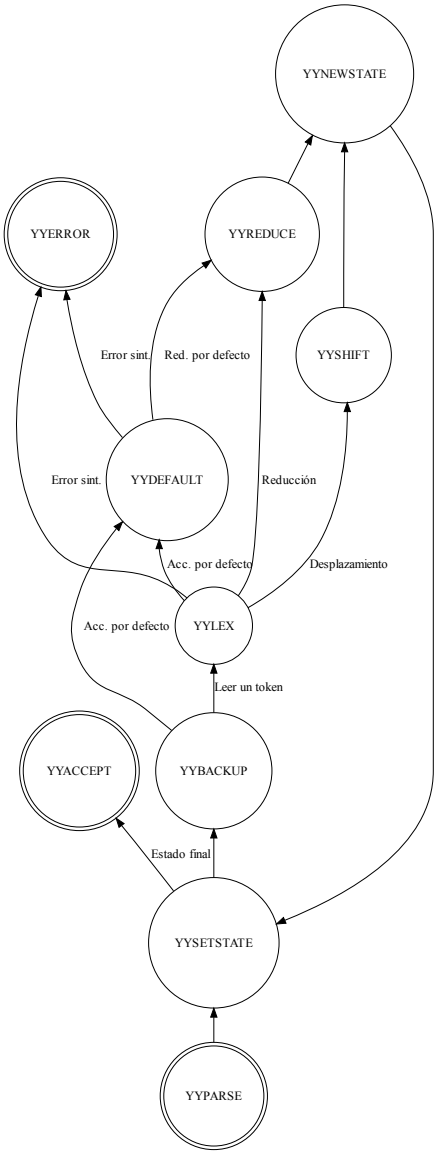


Figura 5.1: Máquina de estados finitos LALR de Bison

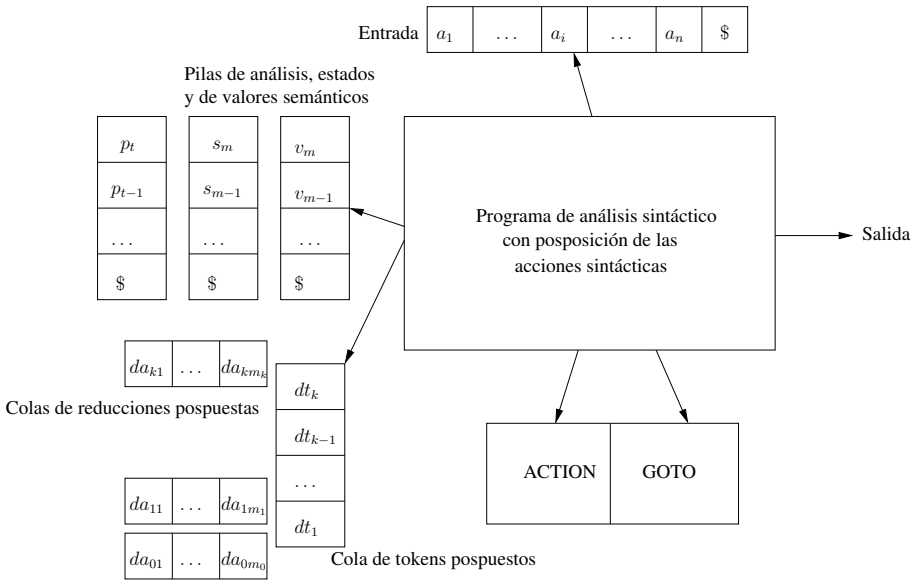


Figura 5.2: Analizador sintáctico LALR(1) con postergación del análisis

5.2. Algoritmo de control con posposición de las acciones del análisis

La máquina de estados finitos descrita en la sección anterior requiere de ciertas modificaciones para permitir un análisis sintáctico pospuesto. A continuación, describimos las estructuras de datos necesarias, además de los cambios a la rutina de control.

5.2.1. Nuevas estructuras de datos

Principalmente, necesitamos almacenar tres conjuntos de datos extra con respecto al algoritmo LALR: la cola de tokens pospuestos, la cola de secuencias de reducciones que ocurren entre estos tokens y una pila de análisis sintáctico. Una vista general del analizador y sus nuevas estructuras puede encontrarse en la Figura 5.2.

Tokens postergados

En general, un analizador sintáctico con k tokens pospuestos puede lograrse por medio de una cola implementada mediante un arreglo circular de tamaño k. El analizador pospuesto siempre tendrá a lo más k tokens, por lo que este tamaño

puede ser definido estáticamente. Hemos definido el valor de k mediante la macro `YYDEFERRAL_LEVEL`, que puede ser redefinida por el usuario, si así lo requiere.

Para el almacenamiento de los tokens postergados, definimos un tipo de datos `YYDEFTOKEN`, cuya estructura contiene toda la información relevante de un token pospuesto: el identificador del token, su valor semántico asociado y, opcionalmente, su ubicación en el archivo de entrada.

Para la implementación en C del arreglo circular utilizamos un arreglo de tamaño $k + 1$, punteros para el comienzo y el final de la cola y una serie de macros que nos permiten operar sobre él. El elemento extra es utilizado para distinguir una cola vacía de una llena.

Reducciones postergadas

Similarmente al almacenamiento de los tokens postergados, utilizamos una cola implementada por medio de un arreglo circular para almacenar las $k - 1$ secuencias de reducciones entre los k tokens postergados. Cada reducción es representada mediante el identificador de la regla mediante la cual ha de reducir y las secuencias son separadas entre sí por medio de un valor de separación que no represente una regla válida.

Debe notarse que, a diferencia de la cola de tokens postergados, estas secuencias no están limitadas en el tamaño que pueden alcanzar, por lo que, al tener un tamaño fijo, puede ser necesario reubicarlas en un espacio de mayor tamaño.

Pila de análisis sintáctico

Además, es necesario introducir una copia de la pila de estados, la cual almacena el estado de la pila antes de la aplicación de la primera secuencia de reducciones y el desplazamiento del primer token postergado. Visto sencillamente, esta pila almacena el estado del analizador que actúa con un retraso de k tokens.

Para esto, introducimos una nueva pila, a la cual llamamos *pila de análisis sintáctico*, implementada de la misma manera que la pila de estados. La diferencia entre la pila de estados y pila de análisis está en que la primera representa el estado del analizador adelantado y la segunda el estado del analizador pospuesto.

5.2.2. Algoritmo de control con postergación del análisis

Una vez que contamos con la infraestructura para la postergación del análisis, estamos en condiciones de adaptar el algoritmo de control. Básicamente, lo que necesitamos es postergar la aplicación de las reducciones sobre la pila de análisis, el desplazamiento de los tokens y la aplicación de las acciones semánticas. La máquina de estados finitos resultante puede apreciarse en la Figura 5.3.

La postergación de la aplicación de las reducciones y de las acciones semánticas es posible cuando, en vez de aplicar una reducción sobre las pilas de estados, valores semánticos y ubicaciones, almacenamos el identificador de la regla correspondiente en la cola de reducciones postergadas y la aplicamos únicamente sobre la pila de estados.

Contando con las estructuras descritas en la sección anterior, la postergación de los desplazamientos es bastante simple. Cuando el algoritmo determina que la acción a realizar corresponde a un desplazamiento, en vez de hacerlo inmediatamente y descartar el token actual, lo almacenamos junto con la información de contexto relevante en nuestra cola de tokens postergados. Si al intentar hacer esto notamos que la cola se encuentra llena, entonces es necesario aplicar la primera secuencia de reducciones pospuestas sobre las pilas de análisis, valores semáticos y ubicaciones, en conjunto con las acciones semánticas correspondientes y luego desplazar el primer token en la cola de tokens postergados, de modo tal de hacer el espacio para el nuevo token postergado.

Una vez que el análisis sintáctico termina correctamente, las colas de tokens y reducciones postergadas son vaciadas, y las acciones semánticas que se encuentran todavía postergadas son aplicadas.

5.3. La rutina de recuperación de errores simples

En esta sección describimos la implementación de la etapa de recuperación simple. Dada la estructura de los algoritmos generados por *Bison*, se hace necesario reestructurar ciertos componentes para permitir la búsqueda de las reparaciones candidatas, por lo que nos referimos también a dichos detalles.

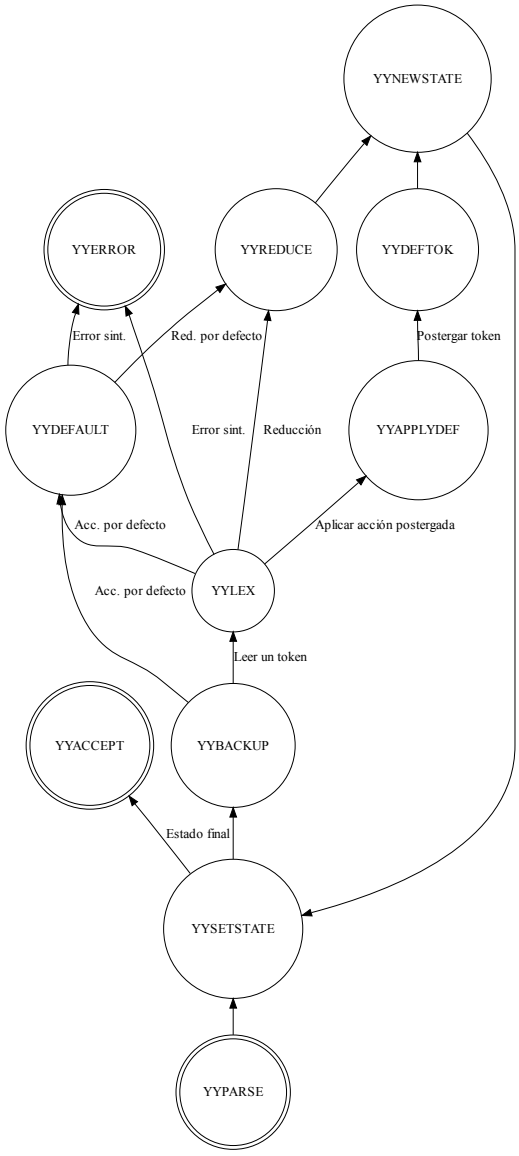


Figura 5.3: Máquina de estados finitos LALR de Bison con postergación del análisis

5.3.1. Detección del error y restauración del estado del analizador

Durante el curso del análisis, un error se detecta cuando, a partir del estado en la parte superior de la pila de estados y el token actual, es imposible aplicar una reducción o desplazar el token, y en ese momento la rutina de recuperación de errores es invocada. Esta rutina comienza restaurando el analizador al estado en el que se encontraba antes de hacer la primera reducción en la cola de secuencias de reducciones postergadas. Esta restauración es posible al copiar los contenidos de la pila de análisis que, como sabemos, tiene contenido postergado, sobre una copia de la pila de estados que se utilizará durante la ejecución de la rutina de recuperación y al considerar como el siguiente token, al primero en la cola de tokens postergados.

5.3.2. Búsqueda de las reparaciones candidatas

Una vez que el analizador sintáctico ha sido restaurado al estado en el que se encontraba antes de analizar el primer token postergado, la rutina de recuperación puede comenzar la búsqueda de reparaciones candidatas. Para esto, implementamos una progresión que simula, desde el primer token postergado hasta el último, el análisis sobre nuestra copia de la pila análisis. Cada prueba de recuperación simple se ejecuta al intentar la inserción, reemplazo o eliminación de un token, como hemos descrito en la sección 3.3.2. Para hacer posible la evaluación de cada una de las reparaciones existentes, implementamos una rutina que simula el análisis sintáctico sobre `TOKENS`, pero permitiéndonos tanto descartar su primer elemento, reemplazarlo por un token arbitrario o insertar uno, lo cual simula la aplicación de uno de los posibles tipos de reparaciones. Esta rutina retorna la cantidad de tokens que la reparación logra analizar por sobre el token de error. Si esta rutina logra analizar por sobre $t_{\min}$ tokens, entonces la corrección se almacena como candidata.

Representación de las reparaciones candidatas

Si la rutina de simulación del análisis sintáctico determina que una reparación es candidata, es necesario almacenar esta reparación para poder seleccionar posteriormente, entre todas las candidatas, la reparación a utilizar. Para almacenar las reparaciones candidatas, buscamos una representación sencilla que, por un lado, no requiera de un espacio considerable en memoria y, por otro lado, permita la aplicación sencilla y efectiva de la corrección elegida una vez que el análisis se reanude.

Decidimos implementar una estructura `YYSIMPLEREPAIR` para almacenar las reparaciones candidatas para errores simples. Esta estructura contiene el tipo de reparación (inserción, eliminación, o reemplazo), el token a utilizar para la reparación, en caso de que esta sea una inserción o un reemplazo, la distancia d entre la posición del primer token postergado y la posición donde la reparación debe aplicarse, la cantidad de tokens t que esta corrección permite al analizador avanzar por sobre el token de error y un campo de validez que es utilizado por la heurística de selección.

Todas las reparaciones candidatas son almacenadas, entonces, en un arreglo de `YYSIMPLEREPAIR`, que luego es utilizado durante la selección de la reparación a utilizar.

Caché de los tokens del escáner léxico

Dada la necesidad de contar con los t_e tokens que siguen al token de error durante la búsqueda de las reparaciones candidatas y al hecho de que la función de análisis léxico que provee la entrada sólo entrega los tokens en demanda (ver la sección 2.4.1 y la Figura 5.1), es necesario contar con un caché de la función `yylex`.

Los analizadores sintácticos generados por *Bison* no cuentan con toda la entrada desde un comienzo, sino que piden el siguiente token a la función de análisis léxico, cada vez que lo requieren. Esto implica, que durante la evaluación de una reparación, no se cuenta con toda la entrada, sino que solamente con el fragmento que se encuentra disponible en la cola de tokens postergados. Sin embargo, la técnica de Burke y Fisher asume que la entrada completa se encuentra disponible en `TOKENS` desde el comienzo del análisis (sección 3.3.2).

Para las evaluaciones que se realicen sobre tokens que se encuentren a una distancia del token de error menor a t_e, es necesario simular el desplazamiento de tokens que todavía no han sido pedidos a `yylex`. Llamar a `yylex` durante una simulación del análisis sintáctico implicaría no contar durante la prosecución del análisis con los tokens que entregue, ya que las llamadas a `yylex`, una vez reanudado el análisis, retornarían nuevos tokens en la entrada.

Para solucionar este problema, decidimos implementar un *mecanismo de caché de la función* `yylex`, mediante el cual podemos almacenar los tokens que `yylex` entregue durante la simulación del análisis sintáctico, para que luego puedan ser utilizados durante el análisis real.

Implementamos este caché utilizando un arreglo de `YYDEFTOKEN` de tamaño t_e. Antes de comenzar la evaluación de las reparaciones, llenamos este caché con t_e tokens provistos por `yylex`, para posibilitar su desplazamiento durante las evaluaciones. Además, envolvimos la macro `YYLEX` que, originalmente, sólo llama a la función `yylex`. Una vez que una reparación es elegida, aplicada y el análisis sintáctico reanudado, `YYLEX` utilizará y descartará uno a uno los tokens en el caché hasta vaciarlo, para luego seguir utilizando los tokens provistos por `yylex` en forma normal.

5.3.3. Selección de la mejor reparación candidata simple

Una vez que se cuenta con un arreglo de reparaciones candidatas, es necesario determinar la mejor reparación posible. Para esto, implementamos la heurística descrita por Burke y Fisher [BF87]. Esta heurística depende fuertemente de las características de la gramática involucrada, por lo que agregamos ciertas directivas a la gramática de entrada de *Bison*, para permitir la especificación de esta información.

A continuación, nos referimos a la implementación de esta heurística, pero sin entrar en los detalles sobre la extensión del formato de entrada de *Bison*. Nos referimos en detalle a las directivas que agregamos y a los cambios necesarios a *Bison* en la sección 5.5.

Heurística de selección

Implementamos la selección de la reparación a ser aplicada, mediante una función `yygetbestrepair`, que recorre el arreglo de reparaciones candidatas reiteradamente, descartando, en base a la información adicional provista por el usuario, las correcciones no deseadas. Para esto utilizamos el campo de validez en `YYSIMPLEREPAIR`, que inicialmente considera a todas las reparaciones como válidas. Dependiendo de la información adicional, cada vez que recorremos el arreglo marcamos como inválidas las correcciones que involucren tokens no preferidos para la corrección o palabras claves. Además, se descartan todas las reparaciones que no son capaces de analizar al menos t_e tokens por sobre el token de error y, en caso de que ninguna logre analizar dicha distancia, sólo se consideran aquellas reparaciones que permiten analizar la mayor cantidad posible de tokens. Finalmente, entre las reparaciones que todavía se consideran válidas, elegimos una de acuerdo al orden de preferencia sugerido por Burke y Fisher: inserción, eliminación, reemplazo.

Debe notarse que, en caso de que el usuario no otorgue información adicional sobre su gramática, la heurística considerará únicamente la cantidad de tokens que cada reparación permita analizar por sobre el token de error y el orden de preferencia sugerido por Burke y Fisher y elegirá arbitrariamente una corrección. Sin embargo, la calidad de las reparaciones, sin esta información adicional, puede ser bastante baja, por lo que es recomendable para los usuarios utilizar siempre las directivas adicionales y proveer la información necesaria.

5.3.4. Prosecución del análisis sintáctico

Una vez elegida una reparación, procedemos a aplicarla y continuar el análisis. Para esto, comenzamos por aplicar análisis sintáctico real sobre los d primeros tokens en la cola de tokens postergados. Luego, aplicamos la corrección correspondiente, copiamos los tokens postergados restantes al final del caché de `yylex`, vaciamos las colas de tokens postergados y de reducciones postergadas y entregamos el análisis a la rutina de análisis principal.

Sabemos que podemos vaciar la cola de tokens postergados, dado que los tokens previos al punto aplicación de la corrección ya fueron desplazados, y los tokens restantes han sido copiados al caché de `yylex`, asegurando que sean usados por la macro `YYLEX` cuando los necesite. De la misma manera, podemos vaciar la cola de secuencias de reducciones postergadas porque sabemos que las secuencias previas al punto de aplicación de la corrección ya fueron aplicadas y que las secuencias posteriores ya no son válidas.

5.4. La rutina de recuperación de errores de ámbito

A continuación, describimos la implementación de la rutina de recuperación de errores de ámbito. Esta rutina es ejecutada cuando la reparación simple falla, es decir, cuando la progresión para la detección de reparaciones candidatas no detecta ninguna o cuando la heurística de selección de reparaciones simples descarta todas las candidatas.

Esta parte de la recuperación depende de la existencia de las secuencias cerradoras de ámbito. Cada secuencia corresponde a una secuencia de uno o más tokens del lenguaje. Estas secuencias son especificadas por el usuario, mediante una directiva que hemos añadido para este propósito (ver sección 5.5).

5.4.1. Búsqueda de la reparación

La búsqueda de una reparación de ámbito reutiliza una parte importante de la infraestructura requerida por la rutina de reparación de errores simples. Antes de comenzar la rutina, volvemos al mismo punto en el cual comienza la reparación simple: la pila de análisis es replicada para poder ser utilizada durante la simulación del análisis y el primer token postergado es marcado como el siguiente en la entrada.

Para facilitar la prueba de cada secuencia cerradora de ámbito, hemos extendido la rutina de simulación del análisis sintáctico, de modo tal que permita la inserción de una secuencia de múltiples tokens. Así, cada prueba de recuperación simple, que involucre una inserción, es, en realidad, un caso particular de una prueba de inserción de una secuencia, la cual está compuesta por solo un token.

Para cada punto de evaluación de la recuperación de ámbito, intentamos la inserción de cada secuencia cerradora de tokens s, hasta encontrar una secuencia que permita a nuestra rutina de simulación del análisis sintáctico proseguir $t_e + |s|$ tokens por sobre el punto de inserción. Tal como lo propone la metodología de Burke y Fisher, la primera inserción que satisfaga esta condición es considerada la reparación de ámbito a utilizar.

5.4.2. Aplicación de la reparación encontrada

Una vez que se encuentra una reparación de ámbito válida, procedemos a aplicarla de forma similar a la aplicación de una reparación simple: se realiza análisis sintáctico real hasta el punto de inserción de los tokens, se aplica la corrección mediante la inserción de los tokens de la secuencia en el caché de `yylex` y los tokens restantes en la cola de tokens postergados son copiados a continuación. Las colas de tokens y secuencias de reducciones son vaciadas y, finalmente, el control es entregado a la rutina principal de análisis sintáctico.

5.5. Extensión del formato de entrada de Bison

Como hemos mencionado en las secciones 5.3.3 y 5.4, para un desempeño óptimo de las rutinas de recuperación de errores implementadas, es necesario que el usuario de *Bison* entregue cierta información adicional sobre la gramática, la cual permite refinar la búsqueda de reparaciones simples y hace posible la búsqueda de

una reparación de ámbito.

En esta sección nos referimos a las distintas directivas que hemos añadido a *Bison* para permitir al usuario entregar esta información de manera simple y transparente, además de los cambios a *Bison* necesarios para procesar esta información y generar su representación dentro del código generado.

5.5.1. Especificación de información para la recuperación simple

La heurística que selecciona la mejor recuperación simple posible, requiere de información adicional para un funcionamiento óptimo –la lista de palabras claves del lenguaje, los tokens preferidos para inserción y eliminación y las sustituciones preferidas. Por esto, hemos extendido el formato de entrada de *Bison* para permitir la especificación, por parte del usuario, de estos datos.

En particular, agregamos las siguientes directivas a bison: `%keyword`, `%prefer` y `%subst`. Cada una de estas directivas puede ser utilizada, opcionalmente, para especificar las palabras claves, los tokens preferidos para inserción y eliminación y las substituciones preferidas, respectivamente.

A continuación nos referimos a los distintos aspectos de la extensión del formato de entrada de *Bison*, notablemente, la modificación de la gramática para los archivos de entrada, la representación interna de la información adicional y la posterior generación de su representación en los analizadores generados.

Extensión de la gramática de entrada

Como es natural imaginar, la descripción de la gramática de los archivos de entrada de *Bison* es, desde luego, una gramática en el formato de *Bison*, la cual es procesada durante la construcción del software para generar el procesador de los archivos de entrada. Este procesador utiliza, a su vez, la herramienta *Flex* para la identificación de los distintos tokens involucrados.

Primero, extendimos la especificación de los tokens de entrada para *Flex*, para añadir los tokens `PERCENT_KEYWORD`, `PERCENT_PREFER` y `PERCENT_SUBST`, los cuales son utilizados cada vez que se logra emparejar la entrada con alguna de las palabras ‘‘`%keyword`’’, ‘‘`%prefer`’’ y ‘‘`%subst`’’, respectivamente.

Con respecto a la gramática de entrada de *Bison*, añadimos las producciones necesarias para analizar sintácticamente la especificación de nuestra información

adicional. Estas producciones pueden verse en el Listado 5.1, en el formato BNF utilizado por *Bison*.

Por claridad, hemos suprimido las acciones semánticas del Listado 5.1. Dichas acciones básicamente llaman a las funciones que hemos añadido a la API de *Bison* para definir las propiedades de nuestro interés en los tokens correspondientes. En la siguiente sección explicamos cómo estas propiedades están representadas internamente, además de las funciones añadidas.

Representación interna de las propiedades extendidas

Bison mantiene internamente una representación de cada uno de los objetos presentes durante el análisis sintáctico, lo cual permite la generación posterior de las tablas de análisis que forman parte del código generado.

En particular, estamos interesados en la representación interna de cada token. *Bison* representa genéricamente cada símbolo gramatical mediante una estructura `symbol`. Esta estructura contiene información relevante para el procesamiento de la gramática, incluyendo el tipo semántico del símbolo, su valor semántico, las reglas de precedencia y asociatividad asociadas, la clase de símbolo y la etiqueta que lo identifica. Parte de estas propiedades son asignadas durante el análisis del archivo de entrada de *Bison* y otras durante el posterior procesamiento de la gramática.

Es en esta estructura donde agregamos tres propiedades que nos permiten especificar toda la información necesaria para la heurística de selección de la reparación simple. La primera propiedad es un campo booleano `is_keyword`, que definimos como verdadero para tokens que estén especificados tras una directiva `%keyword`. De la misma manera, añadimos una propiedad booleana `is_preferred`, que especifica si el símbolo correspondiente ha sido marcado en la gramática como preferido para inserciones o eliminaciones mediante la directiva `%prefer`. Finalmente, añadimos una propiedad para especificar la sustitución preferida para el token, mediante la directiva `%subst`. Esta propiedad es de tipo `symbol_number`, el cual es utilizado internamente por *Bison* para referirse a los identificadores de cada símbolo gramatical.

Listado 5.1: Extensión a la gramática de entrada de *Bison* para la recuperación simple

```
1  grammar_declaration:
2    ...
3  | keywords_declaration
4  | preferred_tokens
5  | substitutions
6  | ...
7  ;
8
9  keywords_declaration:
10  "%keyword" keywords_def.1
11 ;
12
13 keywords_def.1:
14   id
15 | keywords_def.1 id
16 ;
17
18 preferred_tokens:
19   "%prefer" preferred_tok.1
20 ;
21
22 preferred_tok.1:
23   id
24 | preferred_tok.1 id
25 ;
26
27 substitutions:
28   "%subst" id "for" id
29 ;
```

Generación y representación de las propiedades extendidas de los tokens en los analizadores generados

Durante la generación del código, *Bison* manipula una lista con todos los símbolos gramaticales, la cual es usada para expandir las tablas de símbolos en el código generado durante la invocación a *Bison* por parte del usuario. Hemos extendido esta etapa de la generación de código para generar tres tablas extras, las cuales representan las propiedades de nuestro interés.

Hemos sintetizado la información referente a las palabras claves del lenguaje en un arreglo booleano `YYTKEYWORDS`, en donde `YYKEYWORDS[i]` es verdadero si y sólo si el i-ésimo token es una palabra clave del lenguaje, es decir, si el usuario ha listado dicho token junto a la directiva `%keyword`.

Similarmente, se genera un arreglo booleano `YYTPREFERRED`, el cual especifica los tokens preferidos para inserción y eliminación. El i-ésimo elemento en `YYTPREFERRED` es verdadero si y sólo si el i-ésimo token es preferido para inserción o eliminación.

Para la representación de las sustituciones preferidas, *Bison* generará un arreglo `YYTSUBSTS` de tipo entero, en el cual `YYTSUBSTS[i]` tendrá un valor j, en caso de que el usuario especifique la directiva `%subst TOKEN_J for TOKEN_I` en su archivo de gramática.

La generación de los fragmentos de código en la salida de *Bison* es posible gracias a la utilización del lenguaje M4, el cual nos permite insertar macros que hemos definido en nuestra plantilla y que luego son expandidas a los arreglos aquí mencionados.

5.5.2. Especificación de las secuencias cerradoras de ámbito

Para hacer posible la recuperación de ámbito, es necesario que el usuario especifique las secuencias de tokens que representan secuencias cerradoras de ámbito para su lenguaje. Por esto, hemos extendido además el formato de entrada de *Bison* para posibilitar al usuario la especificación de estas secuencias.

A continuación describimos los cambios necesarios a *Bison* para la integración de esta información.

Extensión de la gramática de entrada

Para hacer posible la especificación de las secuencias cerradoras de ámbito, hemos añadido la directiva `%closer` a *Bison*. Cada secuencia es especificada por el usuario

Listado 5.2: Extensión a la gramática de entrada de *Bison* para la recuperación de ámbito

```
1 grammar_declaration:
2     ...
3   | closers_declaration
4   | ...
5 ;
6
7 closers_declaration:
8   %closer" generic_symlist
9 ;
```

de la forma `%closer TOK1 TOK2 ... TOKN`, donde `TOK1 ... TOKN` representa una secuencia cerradora de tokens.

Para permitir la especificación de esta nueva directiva, hemos añadido a la gramática de entrada de *Bison* las producciones en el Listado 5.2. `generic_symlist` es un no terminal ya existente en la gramática de *Bison*, utilizado para especificar una lista genérica de símbolos gramaticales.

Representación interna de las secuencias de cerradores

Representamos en *Bison* cada secuencia de tokens como un arreglo de enteros, donde cada entero corresponde al identificador de un token de la secuencia. Cada vez que una secuencia de cerradores es encontrada en la entrada, uno de estos arreglos es añadido a una lista encadenada simple, donde almacenamos todas las secuencias de tokens internamente.

Representación de las secuencias en los analizadores generados

Cada secuencia es representada en el analizador sintáctico generado, como un arreglo de enteros terminado en `YYUNDEFTOK`, un valor utilizado por *Bison* para especificar un token indefinido. Todos estos arreglos se almacenan en un arreglo de arreglos, para facilitar la progresión entre cada una de las secuencias.

5.6. Uso de la nueva rutina de recuperación de errores

En esta sección presentamos la forma en que la técnica de reparación de errores puede ser utilizada. Discutimos brevemente las interfaces para el usuario, que permiten el uso de esta metodología.

5.6.1. Nuevas interfaces de usuario

La interfaz de usuario de *Bison* corresponde a los distintos parámetros que pueden ser entregados dentro de los archivos de entrada. El usuario, además de especificar la gramática de su lenguaje, debe definir en este archivo algunas funciones y estructuras de datos, que le permitan adaptar a conveniencia la forma del analizador sintáctico a generar. Además, tiene la posibilidad de redefinir algunas macros, lo cual le permite ajustar la sintonía fina del análisis [DS02].

A continuación, describimos las distintas interfaces que hemos añadido, en particular en la forma de *directivas*, *funciones* y *macros*.

Directivas

Para un diagnóstico apropiado de los errores simples y hacer posible la reparación de ámbito, el usuario debe utilizar las siguientes directivas:

`%keyword:` Esta directiva debe ser utilizada para indicar cada una de las palabras claves del lenguaje. Cada ocurrencia de esta directiva debe ir seguida por una o más de las palabras claves.

`%prefer:` Esta directiva permite indicar los tokens que tienen preferencia al momento de realizar una eliminación o inserción. En caso de haber reparaciones candidatas que involucren uno de los tokens especificados junto a esta directiva, las otras reparaciones del mismo tipo serán descartadas.

`%subst:` Mediante esta directiva, el usuario debe indicar las sustituciones que tienen preferencia durante la reparación simple. Cada ocurrencia de esta directiva tiene la forma `%subst TOK1 for TOK2`, donde `TOK1` y `TOK2` son tokens del lenguaje, y `TOK1` es la sustitución preferida para `TOK2`.

`%closer:` Esta directiva permite al usuario indicar cada una de las secuencias de tokens que representan cerradores de ámbito. Esta directiva se debe utilizar de

la forma `%closer TOK1 ... TOKN`, donde `TOK1`, ..., `TOKN` son n tokens que, en ese orden, representan una secuencia cerradora del lenguaje.

La función `yydiagnosis`

La función `yydiagnosis` debe ser provista por el usuario. Esta función será llamada cada vez que se aplique una corrección en la entrada y permitirá al usuario tener conocimiento de la modificación realizada y tomar una decisión con respecto a ésta, ya sea, proseguir el análisis sintáctico, informar mediante un mensaje de advertencia o simplemente abortar el análisis.

El prototipo de esta función es

```
void yydiagnosis (const char const *message);
```

donde `message` representa una versión humanamente leíble y localizada de la reparación aplicada.

Macros

Las siguientes macros representan parámetros del funcionamiento de la metodología y pueden ser redefinidas por el usuario para permitir una sintonía fina del algoritmo de corrección de errores.

`YYDEFERRAL_LEVEL`: Esta macro especifica el nivel de postergación k del análisis sintáctico. Por defecto, está definida como 2. Un mayor nivel de postergación resulta en una mayor capacidad de buscar errores sintácticos, pero implica un mayor uso de memoria y un espacio de búsqueda mayor.

`YYERROR_MIN_THRESHOLD`: Esta macro indica la cantidad de tokens $t_{\min}$ que una corrección debe permitir analizar por sobre el token de error, para que sea considerada una reparación candidata. Por defecto, está definida como 1. Un valor mayor dificulta la búsqueda de reparaciones para los casos en los cuales existen múltiples errores cercanos.

`YYERROR_THRESHOLD`: Esta macro especifica la cantidad de tokens t_e que una corrección simple debe permitir analizar por sobre el token de error, para que la heurística de selección no la descarte. Si no existen correcciones capaces de

analizar por sobre esta cantidad, entonces la heurística seleccionará la corrección que permita analizar la mayor cantidad de tokens posibles, siempre por encima de `YYERROR_MIN_THRESHOLD`. Por defecto, está definida como 5.

6. Resultados

A continuación, revisamos los distintos resultados obtenidos de nuestra implementación de la metodología de Burke-Fisher en *Bison*. En la sección 6.1, mostramos las distintas pruebas que realizamos sobre la metodología, con una serie de lenguajes y código erróneo. Luego, en la sección 6.2 realizamos un análisis de desempeño de nuestra implementación, tanto para el análisis de código correcto como con errores.

6.1. Calidad de la recuperación de errores sintácticos

Para medir la calidad de las recuperaciones, hemos utilizado una métrica basada en la propuesta por Pennello y DeRemer [PD78]. Esta métrica consiste en la clasificación de las reparaciones en tres categorías: *excelentes*, *buenas* y *pobres*. Una reparación se considera *excelente* cuando corresponde a lo que un humano competente haría para corregir el error; *buena*, cuando resulta en un programa razonablemente bueno, sin errores espurios o pasados por alto; y *pobre*, si resulta en uno o más errores o si ocurre una eliminación excesiva de tokens. Además, consideramos la categoría *no corregida*, para aquellos errores que nuestra implementación es incapaz de corregir, y que resultan en la utilización de la recuperación en modo de pánico ya existente en *Bison*.

Con estas categorías, ejecutamos pruebas sobre tres lenguajes distintos. Escribimos una calculadora sencilla para expresiones algebraicas y analizadores sintácticos para los lenguajes Ada y Pascal. Cada una de estas pruebas fueron ejecutadas para distintos grados de postergación k y distintos valores del umbral t_e.

Excelente	Buena	Pobre	No Corregido
13	8	0	3
54.2 %	33.3 %	0 %	12.5 %

Tabla 6.1: Calidad de las reparaciones en una calculadora de escritorio sencilla, $k = 2$, $t_e = 5$

6.1.1. Calculadora de escritorio sencilla

La gramática de la calculadora sencilla está basada en la gramática presentada en la sección 4.9.1 del libro por Aho *et al.* [ALSU06]. Las pruebas realizadas consisten en la ejecución de la calculadora para veinte expresiones matemáticas, con un total de 24 errores sintácticos.

Los resultados obtenidos con $k = 2$ y $t_e = 5$ se encuentran en la Tabla 6.1. La mayoría de las correcciones que se consideran buenas, consisten en la inserción de un entero en un lugar en donde se encuentra un operador de más, cómo en la expresión $5 \times \times 7$, o la inserción de un paréntesis en una expresión en la cual la eliminación del paréntesis sobrante sería preferida, cómo en la expresión $32 - 445 \times (234 - 21)) - 23$. Las correcciones fallidas se deben a la existencia de múltiples errores cercanos, como en la expresión $()34 - 12$. Ante la presencia de al menos dos errores cercanos, es de esperarse que el segundo de ellos dificulte el que alguna corrección del primer error logre analizar por sobre los tokens necesarios, para poder ser considerada candidata.

Una manipulación de t_e puede permitir que esta situación mejore. Definiendo $t_e = 2$, se producen dos correcciones buenas para la expresión $()34 - 12$, antes catalogada como fallida, mientras que el resto se mantiene igual. Para $t_e = 1$, si bien todos los errores son corregidos, se introducen correcciones de baja calidad en cuatro de las pruebas. Finalmente, $t_e = 8$ no produce cambio alguno con respecto a $t_e = 5$.

Con respecto al nivel de postergación, para $k = 5$, la calidad de las reparaciones no varió con respecto a $k = 2$, pese a que tres de las reparaciones buenas variaron a reparaciones de igual calidad, pero detectadas en una posición previa. Deshabilitando la postergación completamente, dos errores, que con $k = 2$ resultaron corregidos excelentemente, no pudieron ser corregidos en lo absoluto, dado que se produjeron en una posición previa al token de error.

Excelente	Buena	Pobre	No Corregido
38	1	2	4
84.4 %	2.2 %	4.4 %	8.9 %

Tabla 6.2: Calidad de las reparaciones en un analizador sintáctico del lenguaje Pascal, $k = 2$, $t_e = 5$

6.1.2. Analizador sintáctico para Pascal

Utilizamos una gramática de Pascal para `yacc`, basada en la presente en el manual de referencia por Doug Cooper [Coo83], para producir un analizador sintáctico para este lenguaje. Ejecutamos un conjunto de veinte programas con un total de 45 errores sintácticos. Doce de estos programas están tomados del artículo por Burke y Fisher [BF87] que presenta originalmente la técnica implementada, mientras que el resto está construido a partir de programas de ejemplo y errores típicos de programación en este lenguaje.

La calidad de las reparaciones, con $k = 2$ y $t_e = 5$, puede apreciarse en la Tabla 6.2. Reduciendo t_e a 2, se redujo considerablemente la calidad de las correcciones en cinco de los programas. Esto se debe a que un valor bajo de t_e permite que un mayor número de reparaciones candidatas sean consideradas, muchas de las cuales introducen errores espurios y producen subsecuentes correcciones espurias que, en conjunto, sólo resultan en un análisis sintáctico incorrecto. Utilizando $t_e = 8$, las reparaciones elegidas no variaron en su calidad.

Para niveles de postergación $k > 2$, no hubo diferencia alguna en las correcciones aplicadas, ya que la mayoría de los errores se encontraban a una distancia del token de error igual o menor a dos tokens. Para un lenguaje como Pascal, $k = 2$ demuestra ser un valor apropiado.

Deshabilitando completamente la postergación del análisis, solo se realizaron correcciones sobre el token de error y siete errores, que con $k = 2$ fueron corregidos, no se corrigieron.

6.1.3. Analizador sintáctico para Ada

Construimos un analizador sintáctico para Ada 95, basado en una gramática para `yacc` disponible en `adaic.org`, para analizar el programa presente en el artículo por Burke y Fisher [BF87], que contiene un total de 32 errores sintácticos. Sobre 19 de ellos se aplicaron correcciones al menos buenas, mientras que doce no fueron

Excelente	Buena	Pobre	No Corregido
17	2	1	12
53.3 %	6.2 %	3.1 %	37.5 %

Tabla 6.3: Calidad de las reparaciones en un analizador sintáctico del lenguaje Ada, $k = 3$, $t_e = 7$

corregidos. El detalle, con un nivel de postergación $k = 2$ y $t_e = 5$, se encuentra en la Tabla 6.3.

Vale la pena mencionar que una de las reparaciones fallidas produjo que la recuperación en modo de pánico descartara un fragmento relativamente extenso de código que contenía otros seis errores sintácticos, sumando esto, siete de los doce errores no corregidos. El primer error descartado se produjo producto de dos tokens erróneos consecutivos. El segundo de estos tokens evitó que se encontrara alguna corrección para el primer error, por lo que la recuperación en modo de pánico resultó invocada. De esto se desprenden dos conclusiones interesantes: la presencia de más de un token erróneo consecutivo puede dificultar considerablemente la búsqueda de reparaciones y, en algunos casos, la recuperación en modo de pánico puede absorber una cantidad considerable de errores, lo que la vuelve bastante subóptima.

Los otros errores no corregidos se produjeron en su mayoría, dada la presencia de múltiples errores consecutivos, tanto simples como de ámbito. En particular, la falta de una implementación recursiva de la técnica de recuperación de ámbitos limita la capacidad de corregir múltiples errores consecutivos.

Para otros valores de k y t_e la calidad de las reparaciones disminuyó considerablemente e incluso se introdujo errores espurios significativamente.

6.2. Análisis de desempeño

Para analizar el desempeño de nuestra implementación, utilizamos nuestro analizador sintáctico para Ada y un archivo de código fuente de 198 kB y alrededor de 5000 líneas de código. Medimos el tiempo total de análisis sintáctico mediante llamadas a la función `clock(3)` de la biblioteca estándar de C, antes y después de la ejecución de la función de análisis. La diferencia entre el resultado de ambas llamadas entrega el tiempo de CPU utilizado por la función en total, el cual, dividido por la macro `CLOCKS_PER_SEC`, permite obtener una aproximación del tiempo de ejecución.

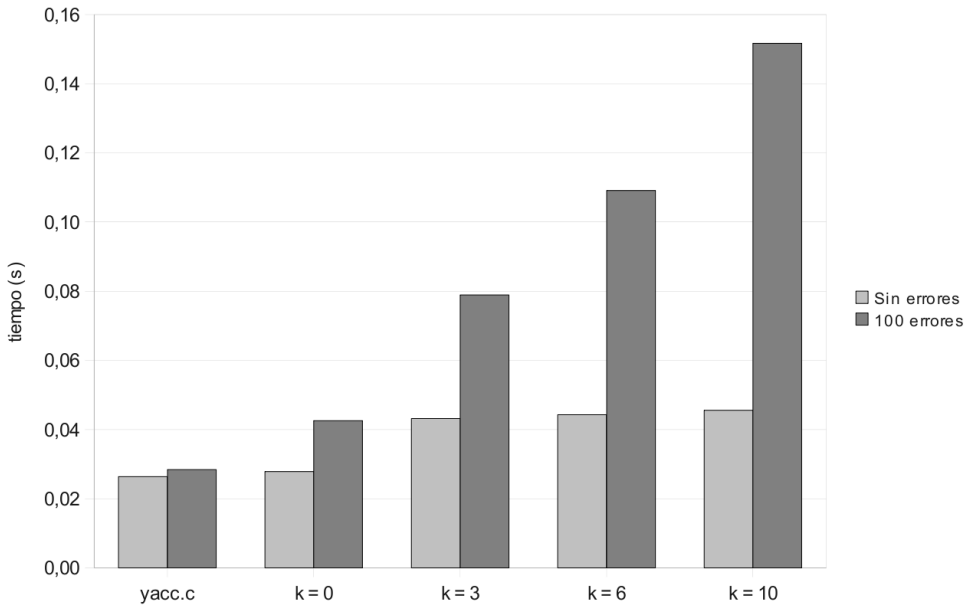


Figura 6.1: Tiempo promedio de análisis en segundos para archivo fuente en Ada de 198 kB sin errores y con 100 errores sintácticos

Para aumentar la precisión, cada prueba se repitió 500 veces y los valores presentados a continuación corresponden a la media aritmética de las repeticiones.

El gráfico en la Figura 6.1 presenta los resultados obtenidos para distintos niveles de postergación del análisis, así como también, para un analizador generado a partir de la plantilla original LALR de *Bison*. De este gráfico se despliegan varias conclusiones interesantes. Primero, el tiempo de análisis para el archivo fuente sin errores mediante el analizador sin postergación ($k = 0$) es, prácticamente, equivalente al tiempo de análisis de la plantilla original. Nuestro analizador sin postergación se encuentra igualmente capacitado para corregir los errores sintácticos, que ocurran sobre el token de error, por lo que es una alternativa interesante para aplicaciones en las cuales no se desee penalizar al código correcto con un mayor tiempo de análisis.

Con respecto a la variación en el tiempo de análisis de código correcto para distintos niveles de postergación, debe notarse que los incrementos son mínimos. Es decir, un grado mayor de postergación no influye considerablemente en el tiempo de análisis para código sin errores.

Con respecto al incremento en el tiempo de análisis para mayores grados de postergación, éstos se deben principalmente a que, a mayor el grado de postergación,

Función	porc. del tiempo
`yyparse`	100 %
`|__yylex`	46.2 %

Tabla 6.4: Desglose del tiempo de ejecución de un analizador sintáctico generado a partir de la plantilla `yacc.c` sobre código sin errores

Función	porc. del tiempo
`yyparse`	100 %
`|__yylex`	54.5 %
`|__yysyntax_error`	9.1 %

Tabla 6.5: Desglose del tiempo de ejecución de un analizador sintáctico generado a partir de la plantilla `yacc.c` sobre código erróneo

mayor es la cantidad de reparaciones evaluadas. La evaluación de cada reparación implica una prueba de análisis sintáctico para determinar la validez de cada reparación, lo cual suma una parte alta de tiempo total del análisis, como mostramos más adelante.

Mediante la herramienta `gprof(1)` de GNU, analizamos la composición del tiempo de ejecución del analizador sintáctico, para poder determinar qué puntos consumen la mayor parte del tiempo de análisis sintáctico. Para obtener un nivel de precisión aceptable, utilizamos dos versiones de un archivo fuente de cerca de 22000 líneas de código: una sin errores y otra con 900 errores sintácticos.

Los resultados para el análisis, tanto del código sin errores, como del código con errores, mediante un analizador sintáctico generado a partir de la plantilla LALR original de *Bison* se encuentran en las Tablas 6.4 y 6.5. De la Tabla 6.4 se desprende que, para un archivo sin errores sintáctico, el 53.8 % del tiempo corresponde al análisis sintáctico propiamente tal, mientras que el 46.2 % restante corresponde a las llamadas al analizador léxico. Ante la presencia de errores (Tabla 6.5), un 54.5 % del tiempo corresponde al análisis léxico, mientras que un 9.1 % corresponde a la generación de los mensajes de error por parte de la función `yysyntax_error`. El análisis sintáctico propiamente tal corresponde tan sólo a un 36.4 %. Esta disminución se debe parcialmente a los fragmentos de la entrada, que son descartados por la rutina de recuperación en modo de pánico.

Como puede verse en la Tabla 6.6, para el análisis sintáctico de código correcto mediante un analizador generado utilizando nuestra plantilla, el porcentaje del tiem-

Función	porc. del tiempo
`yyparse`	100 %
`|__yylex`	16.7 %

Tabla 6.6: Desglose del tiempo de ejecución de un analizador sintáctico generado a partir de la plantilla `yacc-bf.c` sobre código sin errores

Función	porc. del tiempo
`yyparse`	100 %
`|__yymterrorparse`	41.4 %
`|__yylex`	13.8 %
`|__yygetbestrepair`	0.0 %

Tabla 6.7: Desglose del tiempo de ejecución de un analizador sintáctico generado a partir de la plantilla `yacc-bf.c` sobre código erróneo

po que corresponde al análisis sintáctico aumenta considerablemente. Esto queda en evidencia dado que el porcentaje de tiempo dedicado al análisis léxico disminuye de un 46.2 % a un 16.7 %, pese a que la cantidad de tokens en la entrada sigue siendo aproximadamente la misma.

En el caso del análisis de código erróneo mediante un analizador generado a partir de nuestra plantilla, como puede apreciarse en la Tabla 6.7, una parte significativa de la ejecución del analizador corresponde a las pruebas de análisis sintáctico por parte de la función `yymterrorparse`, que corresponden a un 41.4 % del tiempo total. Debe notarse que las llamadas a la heurística `yygetbestrepair` consumen un tiempo despreciable.

6.2.1. Uso de memoria

El uso de memoria se ve incrementado con respecto a los analizadores sintácticos generados en la actualidad por *Bison*. Para medir el incremento, utilizamos el analizador sintáctico para el lenguaje Ada. Como se puede apreciar en la Figura 6.2, entre el analizador original y uno sin postergación de las acciones, hay un incremento en el uso de memoria de alrededor de 20 kB. Este incremento cuenta en parte por las tablas de palabras claves, sustituciones preferidas y tokens preferidos que hemos añadido, además de la maquinaria necesaria para la búsqueda y evaluación de las correcciones. Debe notarse que el tamaño de las tablas es directamente proporcional a la cantidad de tokens del lenguaje y de secuencias cerradoras de tokens.

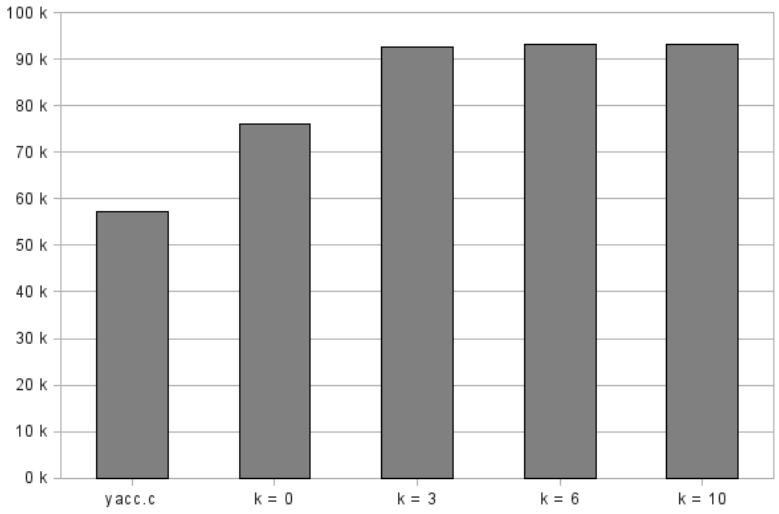


Figura 6.2: Comparación del uso de memoria de distintas versiones de un analizador sintáctico generado

Para analizadores que cuenten con la característica de postergación, se produce un incremento base de alrededor de 16 kB. Este incremento incluye toda la maquinaria necesaria para el tratamiento de las acciones postergadas, además de las colas y pilas que se requieren.

A partir de $k = 1$, no se producen incrementos en el uso de memoria notables para mayores grados de postergación. Esto se debe a que aumentar el nivel de postergación no constituye un incremento en la necesidad de memoria, más allá del aumento de tamaño en las colas y pilas utilizadas, el cual es bastante reducido y no constituye más allá de unos pocos bytes extra por cada una de ellas.

7. Conclusiones

En este trabajo, hemos implementado una metodología para la recuperación automática de errores sintácticos en los analizadores LALR generados por *GNU Bison*. Durante el curso de este trabajo, analizamos distintas técnicas existentes, sus ventajas y desventajas, y finalmente, en base a la literatura reciente y al éxito que esta técnica ha demostrado en la práctica, decidimos implementar la técnica desarrollada por Burke y Fisher. Nuestro trabajo se basa en la plantilla LALR, ya existente en *Bison* y que cuenta con una metodología de recuperación de errores en *modo de pánico*.

Nuestra implementación consiste particularmente en lo que Burke y Fisher describen como *reparación simple* y *reparación de ámbito*. Como nuestros resultados experimentales demuestran, nuestra implementación es capaz de corregir satisfactoriamente la mayoría de los errores sintácticos con los que se encuentre.

Para permitir la corrección de errores sintácticos presentes en puntos previos al token sobre el cual se detecta el error, hemos implementado, además, lo que Burke y Fisher denominan *análisis sintáctico postergado*. Esta técnica, que puede ser vista como "doble análisis sintáctico", permite además separar la verificación de correctitud sintáctica de la aplicación de las acciones semánticas. Si bien nuestros resultados experimentales muestran que el análisis postergado requiere del doble de tiempo para poder llevar a cabo el análisis, es recomendable mantener un grado de postergación de entre 2 ó 3 tokens, ya que esto permite la corrección de una gran parte de los errores sintácticos presentes.

De cualquier forma, esta capacidad puede ser deshabilitada completamente si se requiere un máximo desempeño. De la misma manera, el grado de postergación puede ser redefinido por el usuario, para permitir así una búsqueda de errores sintácticos

más amplia. Sin embargo, nuestros resultados experimentales muestran que un grado de postergación mayor puede significar un aumento del tiempo de búsqueda y, en general, no mejora los resultados considerablemente.

Para un máximo desempeño de la técnica de reparación de errores desarrollada, hemos agregado a *Bison* una serie de directivas, que permiten al usuario especificar información dependiente del lenguaje, como las palabras claves y las secuencias cerradoras de ámbitos. Esta información adicional permite a la metodología corregir errores de acuerdo a la naturaleza particular de cada lenguaje, por lo que si bien, es opcional, recomendamos su utilización.

Código fuente

Es de nuestro especial interés que el trabajo acá presentado sea integrado en futuras versiones de *Bison*. Por esto, hemos mantenido contacto con los mantenedores del proyecto, quienes se han mostrado interesados en adoptar nuestro trabajo. Por esto, a partir del trabajo aquí presentado, es nuestra intención trabajar en conjunto con los desarrolladores de *Bison* para hacer disponibles estas nuevas características a los usuarios de la aplicación en un futuro cercano.

7.1. Limitaciones

En caso de que en la entrada se encuentren tokens mal escritos, la técnica de Burke y Fisher propone la utilización de un algoritmo de búsqueda por similitud, que permita determinar el token correcto. En nuestra implementación, hemos omitido dicha característica, ya que la detección de un token incorrecto solo puede hacerse a nivel del analizador léxico, lo cual escapa del alcance de *Bison* y requiere trabajo en una capa inferior.

La técnica de Burke y Fisher propone, además, la concatenación de dos tokens adyacentes como posible reparación simple. Sin embargo, por las razones ya mencionadas, decidimos no implementar esta característica. Ante la presencia de tokens incorrectos, *Bison* no contará con ellos, por lo que la corrección mediante concatenación de dos tokens sería sólo posible si éstos son tokens válidos del lenguaje y, a la vez, su concatenación resulta en un token válido, lo cual es un caso poco probable.

7.2. Mejoras posibles

Si bien nuestra implementación se desempeña satisfactoriamente para una gran parte de los errores simples y de ámbito, para algunos de los errores más extensos, la metodología de recuperación de errores secundaria pueder resultar en la eliminación de un contexto extenso de la entrada. Consideramos interesante la posibilidad de implementar la metodología de recuperación secundaria propuesta por Burke y Fisher para reemplazar la rutina de recuperación en modo de pánico completamente. De esta manera, podría obtenerse una corrección de errores secundarios más precisa.

Ante la presencia de múltiples errores de ámbito consecutivos, es posible que los errores no sean corregidos apropiadamente. Una implementación recursiva de la metodología de recuperación de errores de ámbito sería ideal para permitir el cierre de múltiples ámbitos abiertos. Sin embargo, una implementación de naturaleza recursiva podría tener serias implicaciones de desempeño. Una alternativa puede ser almacenar completamente el estado de la rutina de recuperación, para permitir que sea restaurada, simulando así la recursividad.

Nuestra implementación requiere que el usuario especifique las secuencias cerradoras del lenguaje para hacer posible la búsqueda de reparaciones de ámbito. La búsqueda de estas secuencias puede ser automatizada completamente. Dada una gramática, Charles [Cha91b] definió formalmente las propiedades que transforman a una secuencia de tokens en una secuencia cerradora de dicha gramática, por lo que es posible, a partir de su definición, desarrollar una metodología que sea capaz de buscar dichas secuencias de manera automática.

Bibliografía

[Ada95] International Organization for Standardization. *Ada 95 Reference Manual. The Language. The Standard Libraries*, January 1995. ANSI/ISO/IEC-8652:1995.

[AG04] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, New York, NY, USA, 2004.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Bac59] John W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *IFIP Congress*, pages 125–131, 1959.

[BBG+60] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Commun. ACM*, 3(5):299–314, 1960.

[BF87] Michael G. Burke and Gerald A. Fisher. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Trans. Program. Lang. Syst.*, 9(2):164–197, 1987.

[Cha91a] Philippe Charles. An LR(k) error diagnosis and recovery method. In *IWPT '91: Proceedings of the 2nd ACL/SIGPARSE International Workshop on Parsing Technologies*, pages 89–99, February 1991.

[Cha91b] Philippe Charles. *A Practical method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery*. PhD thesis, New York University, New York, NY, USA, May 1991.

[Coo83] Doug Cooper. *Standard Pascal: User Reference Manual*. W. W. Norton & Co., Inc., New York, NY, USA, 1983.

[CPRT02] Rafael Corchuelo, José A. Pérez, Antonio Ruiz, and Miguel Toro. Repairing syntax errors in LR parsers. *ACM Trans. Program. Lang. Syst.*, 24(6):698–710, 2002.

[Deg95] Jutta Degener. ANSI C Yacc grammar, 1995. En línea; Consultado el 24 de julio del 2007.

[DP95] Pierpaolo Degano and Corrado Priami. Comparison of syntactic error handling in LR parsers. *Softw. Pract. Exper.*, 25(6):657–679, 1995.

[DS02] Charles Donnelly and Richard M. Stallman. *Bison Manual: Using the YACC-compatible Parser Generator*. GNU Press, Boston, MA, USA, 2002.

[Fre07] Free Software Foundation, Inc. *GNU M4 1.4.10 macro processor*, 2007.

[GJ90] Dick Grune and Ceriel J. H. Jacobs. *Parsing techniques: a practical guide*. Ellis Horwood, Upper Saddle River, NJ, USA, 1990.

[GR75] Susan L. Graham and Steven P. Rhodes. Practical syntactic error recovery. *Commun. ACM*, 18(11):639–650, 1975.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Iro63] E. T. Irons. An error-correcting parse algorithm. *Commun. ACM*, 6(11):669–673, 1963.

[PD78] Thomas J. Pennello and Frank DeRemer. A forward move algorithm for LR error recovery. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 241–254, New York, NY, USA, 1978. ACM Press.

[PEM07] Vern Paxson, Will Estes, and John Millaway. *Lexical Analysis With Flex*, 2007.

[Pop06a] Satya Kiran Popuri. Interactive mode Bison, 2006. En línea; Consultado el 27 de agosto del 2007.

[Pop06b] Satya Kiran Popuri. Understanding C parsers generated by GNU Bison, September 2006. En línea; Consultado el 27 de agosto del 2007.

[Rit93] Dennis M. Ritchie. The development of the C language. *SIGPLAN Not.*, 28(3):201–208, 1993.

[SSS83] Seppo Sippu and Eljas Soisalon-Soininen. A syntax-error-handling technique and its experimental analysis. *ACM Trans. Program. Lang. Syst.*, 5(4):656–679, 1983.

[T+08] Linus Torvalds et al. *Git User's Manual (for version 1.5.3 or newer)*, 2008.

[TA00] David R. Tarditi and Andrew W. Appel. *ML-Yacc User's Manual Version 2.4*, 2000.

ÍNDICE DE FIGURAS

Página

ÍNDICE DE TABLAS

Página

ÍNDICE DE LISTADOS

Página

Índice alfabético