INSTITUTO TECNOLOGICO DE IZTAPALAPA I

INGENIERIA EN SISTEMAS COMPUTACIONALES

Reportes de Apuntes Semanales del

03 AL 07 DE MAYO 2021

Presenta:

PEREZ ARMAS FAUSTO ISAAC

No. De control:

181080037

ASESOR INTERNO:

M.C. ABIEL TOMAS PARRA HERNANDEZ

CIUDAD DE MEXICO **JUNIO/2021**

INDICE

Actividades semana mayo 3-7, 2021

Actividades individuales

1) "Chapter Summary and Perspective" del libro "Engineering a Compiler" de Cooper y Torczon (2012).

2.7 RESUMEN Y PERSPECTIVA DEL CAPÍTULO

El uso generalizado de expresiones regulares para buscar y escanear es una de las historias de éxito de la informática moderna. Estas ideas fueron desarrollando como una parte temprana de la teoría de lenguajes formales y autómatas. Se aplican de forma rutinaria en herramientas que van desde editores de texto hasta filtrado web. Motores a compiladores como un medio de especificar de manera concisa grupos de cadenas que resultan ser lenguajes regulares. Siempre que una colección finita de palabras debe ser reconocidos, los reconocedores basados en dfa merecen una seria consideración.

La teoría de expresiones regulares y autómatas finitos ha desarrollado técnicas que permiten el reconocimiento de lenguajes regulares en tiempo proporcional. a la longitud del flujo de entrada. Técnicas para la derivación automática dudas de res y para la minimización de dfa han permitido la construcción de herramientas robustas que generan reconocedores basados en DFA. Tanto los escáneres generados como los hechos a mano se utilizan en compiladores modernos muy respetados. En cualquier caso, una implementación cuidadosa debe ejecutarse en el tiempo proporcional a la duración del flujo de entrada, con una pequeña sobrecarga por carácter. de exploración crece linealmente con el número de caracteres, y la constante. Los costos son bajos, lo que empuja el análisis léxico del analizador a un el escáner redujo el costo de compilación. El advenimiento de técnicas de análisis eficientes debilitó este argumento, pero la práctica de construir escáneres persiste porque proporciona una clara separación de preocupaciones entre la estructura léxica y estructura sintáctica.

Debido a que la construcción del escáner juega un papel pequeño en la construcción de un compilador real, hemos tratado de que este capítulo sea breve. Por tanto, el capítulo omite muchos teoremas sobre lenguajes regulares y autómatas finitos que el lector ambicioso podría disfrutar. Los muchos buenos textos sobre este tema pueden proporcionar una descripción mucho más profunda tratamiento de autómatas finitos y expresiones regulares, y sus muchas propiedades

Kleene [224] estableció la equivalencia de res y fas. Tanto el Kleene El cierre y el algoritmo de dfa to re llevan su nombre. McNaughton y Yamada mostró una construcción que se relacionares

La construcción que se muestra en este capítulo sigue el modelo del trabajo de Thompson que fue motivado por la implementación de un comando de búsqueda textual para un editor de texto temprano. Johnson describe la primera aplicación de esta tecnología a automatizar la construcción del escáner. La construcción del subconjunto se deriva de Rabin y Scott. El algoritmo de minimización de dfa en la Sección 2.4.4 se debe a Hopcroft Ha encontrado aplicación para muchos problemas diferentes, incluida la detección cuando dos variables de programa siempre tienen el mismo valor

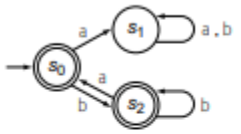
La idea de generar código en lugar de tablas, para producir un código directo escáner, parece tener su origen en el trabajo de Waite y Heuring Informan un factor de cinco de mejora con respecto a las implementaciones basadas en tablas. Ngassam y col. describir experimentos que caracterizan las posibles aceleraciones en escáneres codificados a mano. Varios autores han examinado las compensaciones en implementación del escáner. Jones [208] aboga por la codificación directa, pero defiende un enfoque estructurado para controlar el flujo en lugar del código espagueti que se muestra en la Sección 2.5.2. Brouwer y col. comparar la velocidad de 12 implementaciones de escaneo diferentes; descubrieron un factor de 70 diferencia entre los implementaciones más rápidas y más lentas.

La técnica alternativa de minimización de dfa presentada en la Sección 2.6.2 fue descrito por Brzozowski en 1962 [60]. Varios autores han comparado las técnicas de minimización de dfa y su rendimiento. Muchos los autores han examinado la construcción y minimización de acíclicos

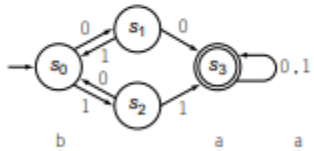
■ EXERCISES

1. Describe informally the languages accepted by the following FAs:

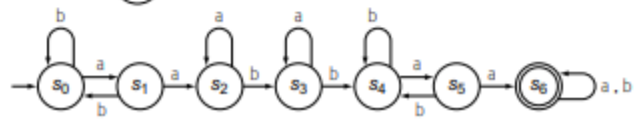
a.



b.



c.



3.7 RESUMEN Y PERSPECTIVA

Casi todos los compiladores contienen un analizador. Durante muchos años, el análisis fue un tema de gran interés. Esto condujo al desarrollo de muchas técnicas para construir analizadores sintácticos eficientes. La familia de gramáticas lr (1) incluye todas las gramáticas libres de contexto que se pueden analizar de forma determinista. Las herramientas producen analizadores sintácticos eficientes con pruebas sólidas propiedades de detección de errores. Esta combinación de características, junto con la amplia disponibilidad de generadores de analizadores sintácticos para lr (1), lalr (1) y slr (1) gramáticas, ha disminuido el interés en otras técnicas de análisis automático como analizadores de precedencia de operadores.

Los analizadores de descendencia recursiva descendente tienen su propio conjunto de ventajas. Ellos son, posiblemente, los analizadores codificados a mano más fáciles de construir. Ellos proveen excelentes oportunidades para detectar y reparar errores de sintaxis. Son eficientes; de hecho, un analizador sintáctico descendente recursivo de arriba hacia abajo bien construido puede ser más rápido que un analizador lr (1) basado en tablas. (El esquema de codificación directa para lr (1) puede superar esta ventaja de velocidad.) En un analizador sintáctico descendente recursivo de arriba hacia abajo, el

El redactor del compilador puede afinar más fácilmente las ambigüedades en el lenguaje fuente. que pueden causar problemas a un analizador lr (1), como un idioma en el que la palabra clave los nombres pueden aparecer como identificadores. Un redactor de compiladores que quiere construir un analizador codificado a mano, por el motivo que sea, se recomienda utilizar el método de descenso recursivo.

Al elegir entre las gramáticas lr (1) y ll (1), la elección se convierte en una de las siguientes: herramientas disponibles. En la práctica, pocas construcciones de lenguaje de programación, si es que hay alguna, caen en la brecha entre las gramáticas lr (1) y ll (1) gramáticas. Por lo tanto, comenzar con un generador de analizador sintáctico disponible es siempre mejor que implementar un generador de analizador desde cero.

Hay disponibles algoritmos de análisis más generales. En la práctica, sin embargo, las restricciones impuestas a las gramáticas libres de contexto por las clases lr (1) y ll (1) no causan problemas para la mayoría de los lenguajes de programación.

■ NOTAS DEL CAPÍTULO

Los primeros compiladores utilizaban analizadores sintácticos codificados a mano La riqueza sintáctica de Algol 60 desafió a los primeros escritores de compiladores. Intentaron una variedad de esquemas para analizar el lenguaje; Randell y Russell ofrecen una visión general fascinante de los métodos utilizados en una variedad de compiladores de Algol Irons fue uno de los primeros en separar la noción de sintaxis de la traducción. Lucas parece haber introducido la noción de descendencia recursiva análisis sintáctico. Conway aplica ideas similares a un eficiente compilador para cobol Las ideas detrás del análisis sintáctico ll y lr aparecieron en la década de 1960. Lewis y Stearns introdujo ll (k) gramáticas; Rosenkrantz y Stearns describieron sus propiedades con mayor profundidad. Foster desarrolló un algoritmo para transformar una gramática en forma ll (1). Wood formalizó la noción de factorización izquierda una gramática y exploró las cuestiones teóricas implicadas en la transformación de una gramática a la forma ll (1)

Knuth expuso la teoría detrás del análisis sintáctico de lr (1). DeRemer y otros desarrollaron técnicas, los algoritmos de construcción de tablas slr y lalr, que hizo que el uso de generadores de analizador lr sea práctico en la memoria limitada computadoras del día. Waite y Goos describen una técnica para eliminar automáticamente las producciones inútiles durante el algoritmo de construcción de la tabla lr (1). Penello sugirió la codificación directa de las tablas. en código ejecutable. Aho y Ullman es una referencia definitiva tanto en el análisis sintáctico de ll como de lr. Bill Waite proporcionó el ejemplo de gramática en ejercicio 3.7.

Aparecieron varios algoritmos para analizar gramáticas arbitrarias libres de contexto en la década de 1960 y principios de la de 1970. Algoritmos de Cocke y Schwartz Younger, Kasami y Earley tenían una complejidad computacional similar. El algoritmo de Earley merece una mención especial debido a su similitud con el algoritmo de construcción de tablas lr (1). Algoritmo de Earley deriva el conjunto de posibles estados de análisis en tiempo de análisis, en lugar de en tiempo de ejecución, donde las técnicas lr (1) los calculan previamente en un generador de analizador sintáctico. A partir de una vista de alto nivel, los algoritmos lr (1) pueden aparecer como una optimización natural del algoritmo de Earley

4. The following grammar is not suitable for a top-down predictive parser. Identify the problem and correct it by rewriting the grammar. Show that your new grammar satisfies the LL(1) condition.

$$L \rightarrow R\,a \mid Q\,ba \qquad R \rightarrow aba \mid caba \mid R\,bc \qquad Q \rightarrow bbc \mid bc$$

5. Consider the following grammar:

$$A \rightarrow B\,a \qquad B \rightarrow dab \mid C\,b \qquad C \rightarrow c\,B \mid A\,c$$

4.6 RESUMEN Y PERSPECTIVA

En los capítulos 2 y 3, vimos que gran parte del trabajo en el frente de un compilador final se puede automatizar. Las expresiones regulares funcionan bien para el análisis léxico. Las gramáticas libres de contexto funcionan bien para el análisis de sintaxis. En este capítulo, Examinamos dos formas de realizar análisis sensibles al contexto: el formalismo gramatical de atributos y un enfoque ad hoc. Para el análisis sensible al contexto, a diferencia del escaneo y el análisis sintáctico, el formalismo no ha desplazado al acercarse.

El enfoque formal, que utiliza gramáticas de atributos, ofrece la esperanza de escribir especificaciones de alto nivel que produzcan ejecutables razonablemente eficientes. Si bien las gramáticas de atributos no son la solución a todos los problemas del análisis sensible al contexto, han encontrado aplicación en varios dominios, que van desde probadores de teoremas hasta análisis de programas. Para problemas en los que el

El flujo de atributos es principalmente local, las gramáticas de atributos funcionan bien. Problemas que se puede formular completamente en términos de un tipo de atributo, ya sea heredado o sintetizados, a menudo producen soluciones limpias e intuitivas cuando se presentan como atributo gramáticas. Cuando el problema de dirigir el flujo de atributos alrededor del árbol con reglas de copia llega a dominar la gramática, probablemente es hora de salir del paradigma funcional de las gramáticas de atributos e introducir un repositorio central de hechos.

La técnica ad hoc, traducción dirigida por sintaxis, integra recortes arbitrarios de código en el analizador y permite al analizador secuenciar las acciones y pasar valores entre ellos. Este enfoque ha sido ampliamente adoptado debido a su flexibilidad y su inclusión en la mayoría de los sistemas generadores de analizadores sintácticos. El enfoque evita los problemas prácticos que surgen de los atributos no locales flujo y de la necesidad de gestionar el almacenamiento de atributos. Los valores fluyen en una dirección junto con la representación interna del analizador de su estado (sintetizados valores para analizadores ascendentes y heredados para analizadores descendentes). Estas Los esquemas utilizan estructuras de datos globales para pasar información en la otra dirección. y para manejar el flujo de atributos no locales.

En la práctica, el redactor del compilador a menudo intenta resolver varios problemas en una vez, como construir una representación intermedia, inferir tipos y Asignación de ubicaciones de almacenamiento. Esto tiende a crear importantes flujos de atributos. en ambas direcciones, empujando al implementador hacia una solución que utiliza algún repositorio central de hechos, como una tabla de símbolos.

La justificación para resolver muchos problemas en una sola pasada suele ser la eficiencia del tiempo de compilación. Sin embargo, resolver los problemas en pasadas separadas puede a menudo producen soluciones que son más fáciles de entender, implementar y mantener.

Este capítulo presentó las ideas detrás de los sistemas de tipos como un ejemplo de tipo de análisis sensible al contexto que debe realizar un compilador. El estudio de la teoría de tipos y el diseño de sistemas de tipos es una actividad académica significativa con una profunda literatura propia. Este capítulo arañó la superficie de la inferencia de tipos y la verificación de tipos, pero un tratamiento más profundo de estos temas está más allá del alcance. alcance de este texto. En la práctica, el redactor del compilador debe estudiar a fondo el sistema de tipos del lenguaje fuente y diseñar la implementación. de inferencia de tipo y comprobación de tipo con cuidado. Los consejos de este capítulo son un comienzo, pero una implementación realista requiere más estudio.

■ NOTAS DEL CAPÍTULO

Los sistemas de tipos han sido una parte integral de los lenguajes de programación desde el compilador original de fortran. Mientras que los primeros sistemas de tipos reflejaban los recursos de la máquina subyacente, pronto niveles más profundos de abstracción aparecieron en sistemas de tipos para lenguajes como Algol 68 y Simula 67.

La teoría de los sistemas de tipos se ha estudiado activamente durante décadas, produciendo una serie de lenguajes que incorporan principios importantes. Éstas incluyen Russell (polimorfismo paramétrico), clu (tipos de datos abstractos), Smalltalk (subtipificación por herencia) y ml (minucioso y tratamiento completo de tipos como objetos de primera clase). Cardelli ha escrito una excelente descripción de los sistemas de tipos. La comunidad apl produjo una serie de artículos clásicos que trataban sobre técnicas para eliminar el tiempo de ejecución cheques

Las gramáticas de atributos, como muchas ideas en informática, fueron propuestas por primera vez por Knuth. La literatura sobre gramáticas de atributos se ha centrado sobre evaluadores, sobre pruebas de circularidad y sobre aplicaciones de gramáticas de atributos. Las gramáticas de atributos han servido como base para varios sistemas exitosos, incluido el compilador Pascal de Intel para el 80286, el sintetizador de programa Cornell y el sintetizador Generador

La traducción ad hoc dirigida por sintaxis siempre ha sido parte del desarrollo de analizadores reales. Irons describió las ideas básicas detrás de la traducción dirigida por sintaxis para separar las acciones de un analizador de la descripción de su sintaxis.

Sin lugar a dudas, se utilizaron las mismas ideas básicas en la precedencia codificada a mano. analizadores sintácticos. El estilo de escribir acciones dirigidas por la sintaxis que describimos fue introducido por Johnson en Yacc. Se ha llevado la misma notación avanzar a sistemas más recientes, incluido el bisonte del proyecto Gnu.

Production	Evaluation Rules
$E_0 \rightarrow E_1 + T$	{ $E_0.nptr \leftarrow mknode(+, E_1.nptr, T.nptr)$ }
$E_0 \rightarrow E_1 - T$	{ $E_0.nptr \leftarrow mknode(-, E_1.nptr, T.nptr)$ }
$E_0 \rightarrow T$	{ $E_0.nptr \leftarrow T.nptr$ }
$T \rightarrow (E)$	{ $T.nptr \leftarrow E.nptr$ }
$T \rightarrow$ id	{ $T.nptr \leftarrow mkleaf(\text{id}, \text{id}.entry)$ }

Actividades en equipo

1) Video de presentación de avance del proyecto final del concepto "Multi-Level Intermediate Representation (MLIR)".

¿ PARA QUE SIRVE ?

TENSOR FLOW

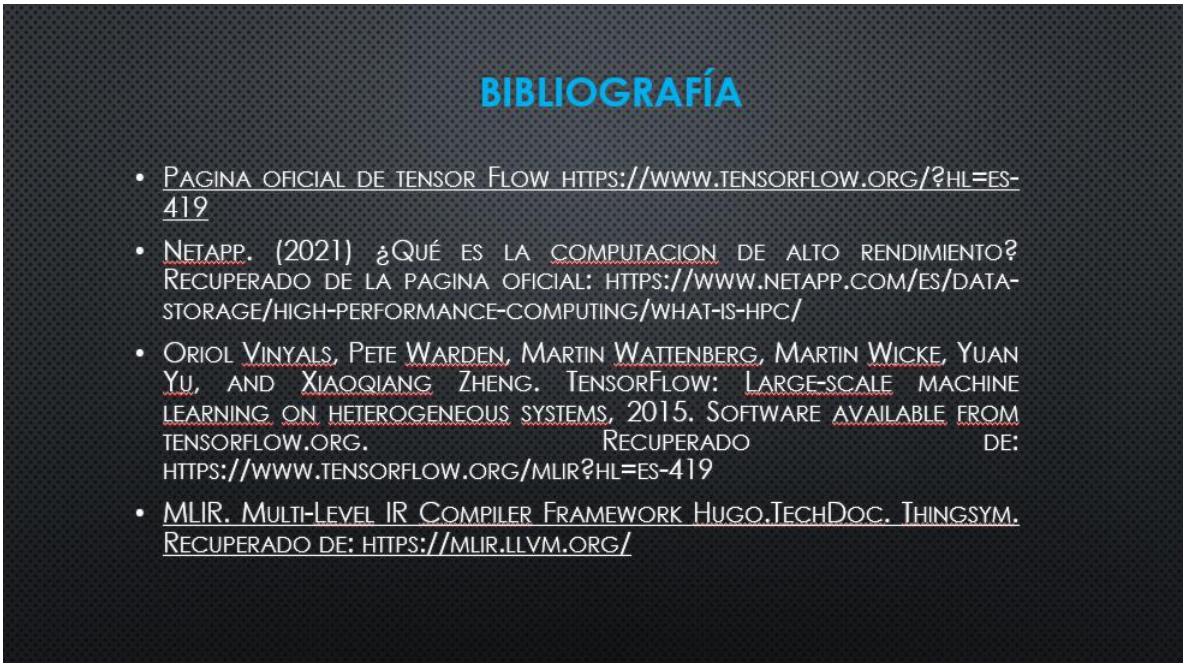
TensorFlow

VENTAJAS DE USAR TENSOR FLOW

- COMPILACIÓN SENCILLA DE MODELOS
- PRODUCCIÓN DE AA SÓLIDO EN CUALQUIER PARTE
- IMPORTANTE EXPERIMENTACIÓN PARA LA INVESTIGACIÓN

¿QUÉ ES HPC?



LIGA DEL VIDEO:

https://youtu.be/a8A6r57KKEw