



Nombre del alumno:	No. Control
Alejandro Arzate Cervantes	171080098
Hernandez Sanchez Ismael	171080133
Medina Piña José Tenatic	171080116
Rojas Martinez Yorman Jesus	161080213

Grupo:

ISC 7AM

Descripción general de la compilación

Un compilador es una herramienta que traduce software escrito en un idioma a otro. Para traducir texto de un idioma a otro, la herramienta debe comprender tanto la forma o sintaxis como el contenido o el significado del idioma de entrada. Debe comprender las reglas que gobiernan la sintaxis y el significado en el lenguaje de salida. Finalmente, necesita un esquema para mapear el contenido del idioma de origen al idioma de destino.

Para mejorar la traducción, un compilador a menudo incluye un optimizador que analiza y reescribe esa forma intermedia.

Los programas de computadora son simplemente secuencias de operaciones abstractas escritas en un lenguaje de programación.

APLICACIONES DE LA TECNOLOGIA DE COMPILADORES

- IMPLEMENTACIÓN DE LENGUAJES DE ALTO NIVEL.
- TRADUCCIONES DE PROGRAMAS.
- GENERACIÓN DE CODIGO MAQUINA PARA LENGUAJES DE ALTO NIVEL.
- HERRAMIENTAS DE PRODUCTIVIDAD DE SOFTWARE.
- PRUEBA DE SOFTWARE.
- DISEÑO DE ARQUITECTURA DE COMPUTADORAS.
- DISEÑO DE DETECCIÓN DE CÓDIGO MALICIOSO PARA NUEVAS ARQUITECTURAS DE COMPUTADORAS.
- INTERPRETES PARA JAVASCRIPT Y FLASH.

También puede realizar ciertas funciones secundarias, como eliminar del programa fuente comentarios y espacios en blanco en forma de caracteres de espacio en blanco, caracteres “pestaña” y de línea nueva. Otra función es relacionar los mensajes de error del compilador



con el código fuente. Por ejemplo, el analizador léxico puede tener localizado el número de caracteres de nueva línea detectados, de modo que se pueda asociar un número de línea con un mensaje de error.

COMPLEJIDAD DE LA TECNOLOGÍA DE COMPILADORES

- Utiliza algoritmos y técnicas de un gran número de áreas de informática.
- Traduce la teoría compleja a la práctica.
- Es el software de sistema más complejo.

SISTEMA DE PROCESAMIENTO DEL LENGUAJE

programa fuente

1.- PREPROCESADOR

Programa fuente final

2.- COMPILADOR

Código ensamblador

3.- ENSAMBLADOR

Objetos en código máquina

4.- ENLAZADOR

Programa final

La estructura de un compilador

Fases de un compilador Analizador léxico flujo de tokens flujo de caracteres Analizador sintáctico Analizador semántico Generador de código intermedio representación intermedia Optimizador de código independiente de la máquina representación intermedia Generador de código

#las diferencia entre el análisis léxico y el análisis sintáctico

Dándonos como principal razón una de simplificación del diseño y mejora de eficiencia de un compilador que va de la mano de la optimización y gestión de un software usando la ingeniería de software. En el diseño del analizador sintáctico, este no debe preocuparse de tener que leer el código fuente o preocuparse con errores de lenguaje inesperados, ya que por eso esta como primera fase el análisis léxico, que funciona como un filtro y le entrega al análisis



sintáctico los tokens necesarios para el funcionamiento de dicho código fuente que puede utilizar en su propia tabla de símbolos como lo vimos en la materia pasada.

ANÁLISIS SINTÁCTICO

El análisis sintáctico convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada. Estos se pueden generar de varias variantes de gramática libre de contexto, es una de las partes de un compilador que transforma su entrada en un árbol de derivación.

Los analizadores no pueden manejar características libres de contexto en lenguajes de programación.

ANÁLISIS SEMÁNTICO

El analizador semántico utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. También recopila información sobre el tipo y la guarda, ya sea en el árbol sintáctico o en la tabla de símbolos, para usarla más tarde durante la generación de código intermedio. Una parte importante del análisis semántico es la comprobación (verificación) de tipos, en donde el compilador verifica que cada operador tenga operando que coincidan.

Es el proceso de relacionar estructuras sintácticas, desde los niveles de frases, cláusulas, oraciones y párrafos hasta el nivel de la escritura en su conjunto, hasta sus significados independientes del lenguaje.

Utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje.

La semántica que no se maneja en el análisis anterior se maneja aquí. La semántica estática de los lenguajes de programación se puede especificar usando gramáticas de atributos.

Almacena la información de tipo en la tabla de símbolos o en el árbol sintáctico.

GENERACION DE CODIGO INTERMEDIO

En el proceso de traducir un programa fuente a código destino, un compilador puede construir una o más representaciones intermedias, las cuales pueden tener una variedad de formas. Los árboles sintácticos son una forma de representación intermedia; por lo general, se utilizan durante el análisis sintáctico y semántico. Después del análisis sintáctico y semántico del programa fuente, muchos compiladores generan un nivel bajo explícito, o una representación intermedia similar al código máquina, que podemos considerar como un programa para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir en la máquina destino.

Muchos compiladores generan un nivel bajo explícito, o una representación intermedia similar al código máquina, que podemos considerar como un programa para una máquina abstracta.



Esta representación intermedia debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir en la máquina destino.

Al convertir el código fuente en un código intermedio, Se puede escribir un optimizador de código independiente de la máquina.

El código intermedio debe ser fácil de producir y fácil de traducir a código de máquina.

OPTIMIZACION DE CODIGO INDEPENDIENTE DE LA MAQUINA

La fase de optimización de código independiente de la máquina trata de mejorar el código intermedio, de manera que se produzca un mejor código destino. Por lo general, mejor significa más rápido, pero pueden lograrse otros objetivos, como un código más corto, o un código de destino que consuma menos poder.

Es el conjunto de fases de un compilador que transforman un fragmento de código en otro fragmento con un comportamiento equivalente y que se ejecuta de forma más eficiente.

La optimización del código consta de un montón de procesos y el porcentaje de mejora depende de los programas

GENERACION DE CODIGO

consiste en código de máquina relocizable o código ensamblador. Las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea. Cada generación de código intermedio podría generar muchas instrucciones de maquina y viceversa. También podría manejar todos los aspectos del código maquina, así como las asignaciones de almacenamiento se toman aquí.

Los tokens, patrones y lexemas

Los componente léxicos o tokens se definen como una secuencia de caracteres con significado sintáctico propio y que son pertenecientes a una categoría léxica (identificador, palabra reservada, literales, operadores o caracteres de puntuación) y estos pueden contener uno o mas lexemas. El lexema es una secuencia de caracteres cuya estructura se corresponde con el patrón de un token y los patrones son la regla que describe los lexemas correspondientes a un token. El patrón es la regla que describe el conjunto de lexemas que pueden representar a un determinado componente léxico en los programas fuente. En otras palabras, es la descripción del componente léxico mediante una regla.

El reconocimiento y especificación de tokens nos habla que se puede realizar mediante un autómata finito, un autómata finito (FA) es una máquina abstracta simple que se utiliza para reconocer patrones dentro de la entrada tomada de algún conjunto de caracteres (o alfabeto) tomando como ejemplo el lenguaje C. El trabajo de un FA es aceptar o rechazar una entrada dependiendo de si el patrón definido por FA ocurre en la entrada.



Herramientas de construcción de compiladores

Al igual que cualquier desarrollador de software, el desarrollador de compiladores puede utilizar para su beneficio los entornos de desarrollo de software modernos que contienen herramientas como editores de lenguaje, depuradores, administradores de versiones, profilers, ambientes seguros de prueba, etcétera. Además de estas herramientas generales para el desarrollo de software, se han creado otras herramientas más especializadas para ayudar a implementar las diversas fases de un compilador. Estas herramientas utilizan lenguajes especializados para especificar e implementar componentes específicos, y muchas utilizan algoritmos bastante sofisticados. Las herramientas más exitosas son las que ocultan los detalles del algoritmo de generación y producen componentes que pueden integrarse con facilidad al resto del compilador. Algunas herramientas de construcción de compiladores de uso común son:

- 1) Generadores de analizadores sintácticos (parsers), que producen de manera automática analizadores sintácticos a partir de una descripción gramatical de un lenguaje de programación.
- 2) Generadores de escáneres, que producen analizadores de léxicos a partir de una descripción de los tokens de un lenguaje utilizando expresiones regulares.
- 3) Motores de traducción orientados a la sintaxis, que producen colecciones de rutinas para recorrer un árbol de análisis sintáctico y generar código intermedio.
- 4) Generadores de generadores de código, que producen un generador de código a partir de una colección de reglas para traducir cada operación del lenguaje intermedio en el lenguaje máquina para una máquina destino.
- 5) Motores de análisis de flujos de datos, que facilitan la recopilación de información de cómo se transmiten los valores de una parte de un programa a cada una de las otras partes. El análisis de los flujos de datos es una parte clave en la optimización de código.
- 6) Kits (conjuntos) de herramientas para la construcción de compiladores, que proporcionan un conjunto integrado de rutinas para construir varias fases de un compilador.
- 7) Lenguajes máquina y ensamblador. Los lenguajes máquina fueron los lenguajes de programación de la primera generación, seguidos de los lenguajes ensambladores. La programación en estos lenguajes requería de mucho tiempo y estaba propensa a errores.

Un paso importante hacia los lenguajes de alto nivel se hizo en la segunda mitad de la década de 1950, con el desarrollo de Fortran para la computación científica, Cobol para el procesamiento de datos de negocios, y Lisp para la computación simbólica. Una de ellas es por generación. Los lenguajes de primera generación son los lenguajes de máquina, los de



segunda generación son los lenguajes ensambladores, y los de tercera generación son los lenguajes de alto nivel, como Fortran, Cobol, Lisp, C, C++, C# y Java. Los lenguajes de cuarta generación son diseñados para aplicaciones específicas como NOMAD para la generación de reportes, SQL para las consultas en bases de datos, y Postscript para el formato de texto. El término lenguaje de quinta generación se aplica a los lenguajes basados en lógica y restricciones, como Prolog y OPS5. Otra de las clasificaciones de los lenguajes utiliza el término imperativo para los lenguajes en los que un programa especifica cómo se va a realizar un cálculo, y declarativo para los lenguajes en los que un programa especifica qué cálculo se va a realizar. Los lenguajes como C, C++, C# y Java son lenguajes imperativos.

8) # Imperativos empleado para expresar mandatos, órdenes, solicitudes, ruegos o deseos.

Modelado en el diseño de compiladores. El diseño de compiladores es una de las fases en las que la teoría ha tenido el mayor impacto sobre la práctica. Entre los modelos que se han encontrado de utilidad se encuentran: autómatas, gramáticas, expresiones regulares, árboles y muchos otros.

Compiladores y arquitectura de computadoras. La tecnología de compiladores ejerce una influencia sobre la arquitectura de computadoras, así como también se ve influenciada por los avances en la arquitectura. Muchas innovaciones modernas en la arquitectura dependen de la capacidad de los compiladores para extraer de los programas fuente las oportunidades de usar con efectividad las capacidades del hardware.

Tecnologías y avances tecnológicos para la creación y adaptación de nuevos modelos teoría y experimentación para la creación o uso de compiladores

#risk y su ventaja en instrucciones para el procesador

Antes se usaban las CISC desbordamiento de los buffers en la administración administración de memoria pero con las nuevas tecnologías la implementación de nuevos modelos computacionales ha sufrido un cambio

El **paralelismo** es una forma de computación en la cual varios cálculos pueden realizarse simultáneamente, basado en el principio de dividir los problemas grandes para obtener varios problemas pequeños, que son posteriormente solucionados en paralelo.

#Multi-procesadores y los beneficios para la programación, distribuir calculo y ""hilos""

Orientación a objetos La orientación a objetos se introdujo por primera vez en Simula en 1967, y se ha incorporado en lenguajes como Smalltalk, C++, C# y Java. Las ideas claves de la orientación a objetos son:

- La abstracción de datos
- La herencia de propiedades



Entornos. La asociación de nombres con ubicaciones en memoria y después con los valores puede describirse en términos de entornos, los cuales asignan los nombres a las ubicaciones en memoria, y los estados, que asignan las ubicaciones a sus valores.

Los lenguajes de programación están diseñados para ofrecer expresividad, concisión y claridad. Los lenguajes naturales permiten la ambigüedad. Los lenguajes de programación están diseñados para evitar la ambigüedad; un programa ambiguo no tiene sentido.

Los lenguajes de programación están, en general, diseñados para permitir que los humanos expresen cálculos como secuencias de operaciones. Los procesadores de computadora, en lo sucesivo denominados procesadores, microprocesadores o máquinas, están diseñados para ejecutar secuencias de operaciones. Las operaciones que implementa un procesador son, en su mayor parte, a un nivel de abstracción mucho más bajo que las especificadas en un lenguaje de programación.

La herramienta que realiza tales traducciones se llama compilador. El compilador toma como entrada un programa escrito en algún lenguaje y produce como salida un programa equivalente.

Los lenguajes de "fuente" típicos pueden ser c, c ++, fortran, Java o ml.

Algunos compiladores producen un programa de destino escrito en un lenguaje de programación orientado a humanos en lugar del lenguaje ensamblador de alguna computadora.

Los programas que producen estos compiladores requieren una traducción adicional antes de que puedan ejecutarse directamente en una computadora. Muchos compiladores de investigación producen programas en C como resultado. Debido a que los compiladores para C están disponibles en la mayoría de las computadoras, esto hace que el programa de destino sea ejecutable en todos esos sistemas, a costa de una compilación adicional para el destino final. Los compiladores que se dirigen a lenguajes de programación en lugar del conjunto de instrucciones de una computadora a menudo se denominan traductores de fuente a fuente.

Algunos lenguajes, como Perl, Scheme y apl, se implementan con más frecuencia con intérpretes que con compiladores. Algunos idiomas adoptan esquemas de traducción que incluyen tanto la compilación como la interpretación. Java se compila a partir del código fuente en un formato llamado bytecode, una representación compacta destinada a reducir los tiempos de descarga de la máquina virtual. (Una máquina virtual es un simulador de algún procesador). Es un intérprete del conjunto de instrucciones de esa máquina.

Un buen compilador combina ideas de la teoría del lenguaje formal, del estudio de algoritmos, de la inteligencia artificial, del diseño de sistemas, de la arquitectura informática y de la teoría de los lenguajes de programación y las aplica al problema de traducir un programa. Un compilador reúne algoritmos codiciosos, técnicas heurísticas, algoritmos de gráficos, programación dinámica, dfas y nfas, algoritmos de punto fijo, sincronización y localidad, asignación y denominación, y gestión de canalizaciones.



Para ubicar esta actividad en un marco ordenado, la mayoría de los compiladores se organizan en tres fases principales: un front-end, un optimizador y un back-end.

Front End

La interfaz determina si el código de entrada está bien formado, en términos tanto de sintaxis como de semántica. Si encuentra que el código es válido, crea una representación del código en la representación intermedia del compilador; Si no, informa al usuario con mensajes de error de diagnóstico para identificar los problemas con el código.

Optimizador

Traduce un programa dentro de otro, con el objetivo de producir un programa ir que se ejecute de manera eficiente. Los optimizadores analizan programas para obtener conocimientos sobre su comportamiento en tiempo de ejecución y luego utilizan ese conocimiento para transformar el código y mejorar su comportamiento.

Back End

El back-end del compilador atraviesa la forma ir del código y emite código para la máquina de destino. Selecciona las operaciones de la máquina objetivo para implementar cada operación de infrarrojos. Elige un orden en el que se ejecutarán las operaciones eficientemente. Decide qué valores residirán en los registros y qué valores residirán en la memoria e inserta código para hacer cumplir esas decisiones



>>Actividades semana 5 (Oct 19-23, 2020)<<

Tabla de transición

Tabla de transición de estados es una tabla que muestra qué estado se moverá un autómata finito dado

Diferencias entre NFA >><< DFA

Un NFA es un autómata Finito no determinista. No determinista significa que puede a la transición, y estar en varios estados a la vez.

Un DFA es un Autómata Finito Determinista. Determinista significa que solo puede estar en, y la transición a un estado en un momento.

Para cada AFND, existe un AFD que acepta el mismo lenguaje.

El estado de inicio de un DFA podría corresponder a (el alfabeto que pertenece a q_0) $\{q_0\}$ y podría ser representado por $[q_0]$

inicializado en la forma $\delta([q_0].a)$, el nuevo estado del DFA será construido bajo demanda

cada sub-conjunto del estado NFA es un posible estado de DFA

todos los estado del DFA contiene estado final como un miembro podría ser el estado final del DFA

La generación de un nuevo nodo para generar transiciones cíclicas

El estado de inicio de un DFA podría corresponder a (el alfabeto que pertenece a q_0) $\{q_0\}$ y podría ser representado por $[q_0]$

Inicializado en la forma $\delta([q_0].a)$, el nuevo estado del DFA será construido bajo demanda

Cada sub-conjunto del estado NFA es un posible estado de DFA

Todos los estado del DFA contiene estado final como un miembro podría ser el estado final del DFA

Los diagramas de transición son DFA generalizados con la siguiente:

Los bordes pueden ser etiquetados por un símbolo, un conjunto o una definición regular algunos estado de aceptación pueden ser identificados como estados retractantes (lo que indica que el lexema no incluye el nodo que nos llevó al estado de aceptación)



Cada estado de aceptación tiene una acción adjunta a la cual es el estado cuando se alcanza ese estado típicamente tal acción devuelve un token y su valor de atributo los diagramas de transición no están destinados a maquinaria.

La forma en que se combinan las palabras y símbolos para formar programas ejecutables puede parecer confusa y casi antojadiza. Sin embargo, la escritura de programas computacionales se rige por un reducido conjunto de reglas gramaticales. Este conjunto de reglas se denomina la *sintaxis* del lenguaje de programación. Por esta razón, también se habla de reglas sintácticas como sinónimo de reglas gramaticales.

Los analizadores son generados para una gramática en particular, esta es utilizada en general para lenguajes de programación. Los lenguajes de programación pueden ser generados por cierta clase de gramática.

DIAGRAMAS DE TRANSICIÓN

representan las acciones que tienen lugar cuando el analizador léxico es llamado por el analizador sintáctico para obtener el siguiente componente léxico.

Se utiliza un diagrama de transición para localizar la información sobre los caracteres que se detectan a medida que el apuntador delantero examina la entrada. Esto se hace cambiando de posición en el diagrama según se leen los caracteres.

Los dos elementos principales en estos diagramas son los estados y las posibles transiciones entre ellos.

- El estado de un componente o sistema representa algún comportamiento que es observable externamente y que perdura durante un periodo de tiempo finito. Viene dado por el valor de uno o varios atributos que lo caracterizan en un momento dado.
- Una transición es un cambio de estado producido por un evento y refleja los posibles caminos para llegar a un estado final desde un estado inicial.

Desde un estado pueden surgir varias transiciones en función del evento que desencadena el cambio de estado, teniendo en cuenta que, las transiciones que provienen del mismo estado no pueden tener el mismo evento, salvo que exista alguna condición que se aplique al evento.

Un sistema solo puede tener un estado inicial, que se representa mediante una transición sin etiquetar al primer estado normal del diagrama. Pueden existir varias transiciones desde el estado inicial, pero deben tener asociadas condiciones, de



manera que solo una de ellas sea la responsable de iniciar el flujo. En ningún caso puede haber una transición dirigida al estado inicial.

El estado final representa que un componente ha dejado de tener cualquier interacción o actividad. No se permiten transiciones que partan del estado final. Puede haber varios estados finales en un diagrama, ya que es posible concluir el ciclo de vida de un componente desde distintos estados y mediante diferentes eventos, pero dichos estados son mutuamente excluyentes, es decir, sólo uno de ellos puede ocurrir durante una ejecución del sistema.

LEX, UN GENERADOR DE ANALIZADORES LÉXICOS

Es un generador de programas diseñado para el proceso léxico de cadenas de caracteres del código fuente de un programa. El programa acepta una especificación, orientada a resolver un problema de alto nivel para comparar literales de caracteres, y produce un programa C que reconoce expresiones regulares.

Internamente Lex va a actuar como un autómata que localizará las expresiones regulares que le describamos, y una vez reconocida la cadena representada por dicha expresión regular, ejecutará el código asociado a esa regla.

Se caracteriza por tener un lenguaje propio para describir las expresiones regulares provenientes de los FA y generar realciones de esas mismas con lo que se esta ingresando.

ESTRUCTURA DE LEX

la estructura general de LEX es:

{definiciones}

%%

{reglas}

%%

{rutinas del usuario}

De estas tres secciones, sólo la segunda es obligatoria, y cualquiera de ellas

puede estar vacía. Si no se llegase a utilizar alguna de estas secciones esta es reemplazada por %%.

- **La sección de declaraciones** incluye declaraciones de variables, constantes



y definiciones regulares, que constituyen una manera cómoda de utilizar expresiones regulares largas en la sección de reglas; por ejemplo: letra [A-Za-z]

- **La sección de reglas** especifica los patrones a reconocer y las acciones asociadas a éstos, de forma similar a la que utiliza awk: patrón {acciones en C}
- **La sección de rutinas** permite definir funciones auxiliares en C, incluida la función main(). Por defecto, lex proporciona un main() que simplemente llama a la función yylex().

VARIABLES, METODOS Y MACROS DE LEX

Variables:

Variable	Tipo	Descripción
yytext	char * o char []	Contiene la cadena de texto del fichero de entrada que ha encajado con la expresión regular descrita en la regla.
yylength	int	Longitud de yytext. yylength = strlen (yytext).
yyin	FILE *	Referencia al fichero de entrada.
yyval	struct	Contienen la estructura de datos de la pila con la que trabaja YACC. Sirve para el intercambio de información entre ambas herramientas.
yylval		

Métodos:

Método	Descripción
yylex ()	Invoca al Analizador Léxico, el comienzo del procesamiento.
yyomore ()	Añade el yytext actual al siguiente reconocido.
yyless (n)	Devuelve los n últimos caracteres de la cadena yytext a la entrada.
yyerror ()	Se invoca automáticamente cuando no se puede aplicar ninguna regla.
yywrap ()	Se invoca automáticamente al encontrar el final del fichero de entrada.

Macros:

Nombre	Descripción
ECHO	Escribe yytext en la salida estandar. ECHO = printf ("%s", yytext)
REJECT	Rechaza la aplicación de la regla. Pone yytext de nuevo en la entrada y busca otra regla donde encajar la entrada. REJECT = yyless (yylength) + 'Otra regla'
BEGIN	Fija nuevas condiciones para las reglas. (Ver Condiciones sobre Reglas).
END	Elimina condiciones para las reglas. (Ver Condiciones sobre Reglas).



LEX Y LA IMPLEMENTACIÓN DEL LENGUAJE C EN EL GENERADOR

Muchas veces es útil poder incluir código en C dentro de un programa en lex, para apoyar el trabajo del analizador. Suele ser normal, por ejemplo, declarar una estructura de datos que contenga detalles sobre cada lexema encontrado o incluir nuestra propia función `main()`.

Puede insertarse código C en un fuente Lex en varias partes:

- En la sección de declaraciones, entre una línea `%{` y una línea `%}` (nótese que no es `%}`). Este código será externo y se situará antes de la función `yylex()` en el programa `lex.yy.c`.
- En la sección de declaraciones, cualquier línea que comience por un espacio en blanco se considerará código C y será también externo y anterior a `yylex()`.
- En la sección de reglas, y antes de que empiece la primera regla, toda línea que comience por un espacio en blanco se considerará código C que será interno a la función `yylex()`. Normalmente no es necesario incluir código C de esta manera.
- En la sección de reglas, las acciones en C de cada regla serán parte del código de `yylex()`.
- Toda la sección de rutinas es código C, externo y posterior a la función `yylex()`. Puede utilizarse para cualquier cosa, pero en particular, puede usarse para definir una función `main()` distinta a la que se genera por defecto.



EJEMPLO BÁSICO DE LEX (RECONOCIMIENTO DE PALABRAS Y NÚMEROS)

```
numeros    [0-9]
letras     [A-Za-z]
separador[ \t\n]
espblanco{delim}+

%%
{espnblanco}      { /* se ignoran espacios en blanco */ }
{numero}+ { printf("%s", yytext); printf(" = numero\n"); }
{letra}+ { printf("%s", yytext); printf(" = palabra\n"); }
%%

main()
{
    yylex();
}
```

QUE ES EL ANÁLISIS O EL ANALIZADOR SINTACTICO

ACTIVIDADES DE UN ANALIZADOR DE LA GRAMÁTICA DE UN LENGUAJE DE PROGRAMACIÓN

- Verifica que la cadena de tokens de un programa se genera a partir de dicha gramática.
- Reporta los errores de sintaxis en el programa.
- Podría ser generado a mano o en automático se basa en gramática libre de contexto.
- Cuando es necesario llama al analizador léxico para proporcionarle un token.

GRAMÁTICAS LIBRE DE CONTEXTO

estas gramáticas son suficientemente simples como para permitir el diseño de eficientes algoritmos de análisis sintáctico que, para una cadena de caracteres dada.

Se denota por:



Una gramática libre de contexto se define con

$G = (V, T, P, S)$ donde:

- V es un conjunto de variables
- T es un conjunto de terminales
- P es un conjunto finito de producciones de la forma $A \rightarrow \alpha$, donde A es una variable y $\alpha \in (V \cup T)^*$
- S es una variable designada llamada el símbolo inicial

LENGUAJES LIBRE DE CONTEXTO

Son los lenguajes formales que engloban a los lenguajes regulares y constituyen los mecanismos de representación y reconocimiento de los lenguajes de programación desde el punto de vista sintáctico.

un lenguaje es libre de contexto (LLC) si y sólo si puede ser reconocido por una gramática libre de contexto (GLC), es aquel cuyas cadenas pueden ser reconocidas por autómatas de pila.

Los LLC tienen vital importancia en el diseño e implementación de lenguajes de programación pues sus implementaciones informáticas suelen constituir los núcleos de los parsers o analizadores sintácticos y son parte del análisis semántico.

ARBOL DE DERIVACIÓN

Un árbol es un conjunto de puntos, llamados nodos, unidos por líneas, llamadas arcos. Un arco conecta dos nodos distintos. Para ser un árbol un conjunto de nodos y arcos debe satisfacer ciertas propiedades.

Los nodos internos del árbol son no terminales y las hojas terminales, el rendimiento del árbol depende de la lectura de izquierda a derecha de todas las etiquetas.

El árbol de derivación tiene las siguientes propiedades:

- el nodo raíz está rotulado con el símbolo distinguido de la gramática;
- cada hoja corresponde a un símbolo terminal o un símbolo no terminal;
- cada nodo interior corresponde a un símbolo no terminal.

GRAMÁTICA AMBIGUA

una gramática ambigua es un Gramática libre del contexto para la que existe una cadena que puede tener más de una derivación a la izquierda, mientras una gramática no ambigua es una



Gramática libre del contexto para la que cada cadena válida tiene una única derivación a la izquierda.

La ambigüedad en una gramática puede eliminarse creando nuevas producciones que definan el mismo conjunto de cadenas (strings) que las originales.

La ambigüedad puede eliminarse definiendo normas o reglas específicas durante la construcción de la tabla de análisis para resolver los casos al momento que se presentan.



ABCD

AUTÓMATA DE EMPUJE

Los autómatas pushdown se utilizan en teorías sobre lo que pueden calcular las máquinas. Son más capaces que las máquinas de estados finitos pero menos capaces que las máquinas de Turing. Los autómatas pushdown deterministas pueden reconocer todos los lenguajes deterministas libres de contexto, mientras que los no deterministas pueden reconocer todos los lenguajes libres de contexto, siendo el primero de uso frecuente en el diseño de analizadores sintácticos.

ESTRUCTURA

Una PDA se define formalmente como una tupla de 7:

$M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ donde

- Q es un conjunto finito de *estados*
- Σ es un conjunto finito que se llama *alfabeto de entrada*
- Γ es un conjunto finito que se llama *alfabeto de pila*
- δ es un subconjunto finito de la *relación de transición* $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times Q \times \Gamma^*$
- $q_0 \in Q$ es el *estado de inicio*
- $Z \in \Gamma$ es el *símbolo de pila inicial*
- $F \subseteq Q$ es el conjunto de *estados de aceptación*



>>[Actividades semana 6 \(Oct 26-30, 2020\)](#)Tarea<<

En este video se discute sobre el autómata Pushdown de gramática libre de contexto y luego el análisis arriba hacia abajo y de abajo hacia arriba

Pushdown tiene un conjunto finito de estados Q , tiene un alfabeto de entrada tiene una coma de alfabeto de pila hay un estado de pila de inicio z y un conjunto de estados finales F

En el conjunto Q de un estado q en un símbolo de entrada y los símbolos de pila agregan en la parte superior de stack pueden moverse al estado $P1$ o $P2$ o es $p1, p3$ y en ese proceso quita la parte superior del símbolo de la pila y lo reemplaza con gama 1, gama 2 cualquiera de estos dependiendo de en que estado se mueva este es el símbolo de entrada por uno

A PDA M is a system $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, where

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $q_0 \in Q$ is the start state
- $z_0 \in \Gamma$ is the start symbol on stack (initialization)
- $F \subseteq Q$ is the set of final states
- δ is the transition function, $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$ to finite subsets of $Q \times \Gamma^*$

A typical entry of δ is given by

$$\delta(q, a, z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_m, \gamma_m)\}$$

The PDA in state q , with input symbol a and top-of-stack symbol z , can enter any of the states p_i , replace the symbol z by the string γ_i , and advance the input head by one symbol.

El algoritmo de Cocke-Younger-Kasami (CYK)

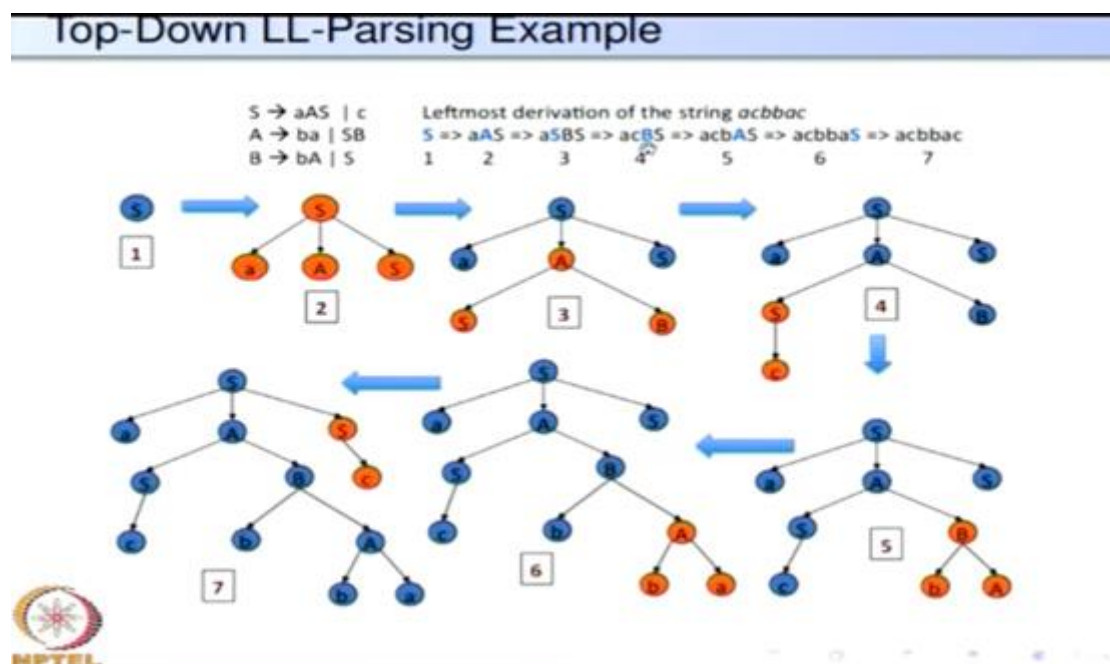
El algoritmo determina si una cadena puede ser generada por una gramática libre de contexto y, si es posible, cómo puede ser generada. Este proceso es conocido como análisis sintáctico de la cadena. El algoritmo es un ejemplo de programación dinámica.

La versión estándar de CYK reconoce lenguajes definidos por una gramática libre de contexto escrita en la forma normal de Chomsky (CNF). Cualquier gramática libre de contexto puede ser convertida a CNF sin mucha dificultad, CYK puede usarse para reconocer cualquier lenguaje libre de contexto. Es posible extender el algoritmo CYK para que trabaje sobre algunas gramáticas libre de contexto no escritas como CNF. Esto puede hacerse para mejorar la ejecución, aunque hace el algoritmo más difícil de entender.

Los subconjuntos de lenguajes libres de contexto suelen requerir $O(n)$ tiempo

Análisis predictivo utilizando gramáticas LL(1)(análisis de método de arriba hacia abajo)

Reduzca el análisis sintáctico utilizando la gramática LA(1) (método de análisis sintáctico de abajo hacia arriba



Predecir los productos en esto es aplicable en este momento. Entonces hay información adicional disponible para eso, mire esa información más adelante, pero en este punto, ya sabemos que la elección que se ha hecho va a ser solo entonces

Entonces tenemos dos no terminales *B* más, y sí, saber, para que eso Oh, sí bueno a esta cosa en particular va a ver que se ha completado en este paso y este sí así es el resto de misa para que se expanda por van a ser *a* y finalmente el sobrante no terminal se expande porque va a ver así como se puede ver la reconstrucción pasada y la derivación mas a la izquierda están sincronizados, así que están sincronizado así que sabemos qué no terminal en particular se expande en el árbol de análisis en el siguiente paso.

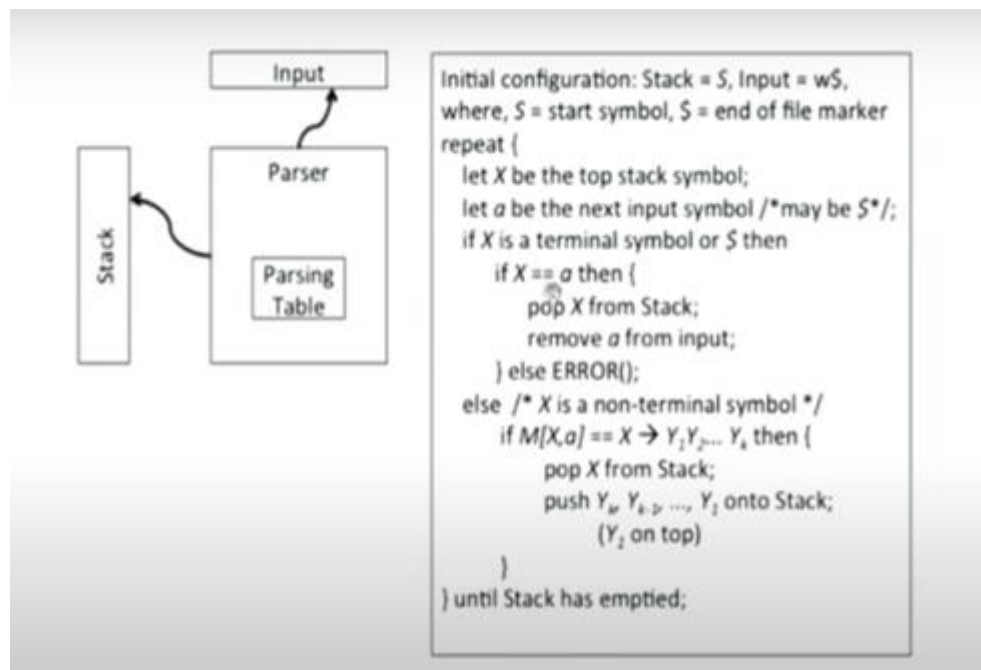
Entonces el análisis de arriba hacia abajo usando pasos predictivos eleva la derivación más a la izquierda de la cadena mientras se construyen las tres primeras, así que comenzamos desde el símbolo de inicio y predecimos la producción que se usa en la derivación, necesitamos más información y eso se conoce como tabla de análisis, que se construye sin conexión y se detiene, por lo que vamos a estudiar cómo la partícula es efectiva en la

La tabla de análisis que se construye fuera de línea y se almacena. Entonces vamos a estudiar cómo se construye la tabla de análisis sintáctico. Las próximas entrevistas de producción en la derivación se determinan observando el siguiente símbolo y también la tabla del analizador así que esta combinación te dice exactamente qué producción en particular se utilizará y el símbolo *X* que vemos se llama cuando miramos hacia adelante así que al imponer restricciones a la gramática, nos aseguramos de que no haya luego una reducción en cualquier tipo de tabla de personas, por lo que veremos que si hay más de una producción en cualquier trama de la tabla de cierre, entonces no podemos decidir qué producción usar a Entonces, en el momento de la construcción de la tabla de análisis, hay dos Producciones elegibles para colocarse en el mismo tipo de tabla de personas.



Entonces se declara que la gramática no es apta para el derecho de así que sigamos adelante y veamos cómo funciona exactamente el algoritmo

Así que repite que X sea el símbolo de la pila superior para que en algún momento para empezar sea sí, y luego pueda convertirse en algún otro símbolo, deja que a sea el siguiente símbolo de podría ser el final del archivo en así que si la parte superior de la pila X es un símbolo de terminal o el símbolo de fin de archivo dólar y es igual al símbolo de entrada a , entonces obviamente es hora de parchear el conjunto que el repetidor puso en el autómata



Luego puedes convertirte en otros símbolos, deja el siguiente símbolo de entrada de un Podría ser un dólar de Entonces la parte superior de la pila X es un símbolo de terminal o el símbolo de fin de archivo dólar y es igual al símbolo de entrada a , entonces obviamente diseñado para tocar el pero si nos pusieron en autómata también hace esto así que siempre que el símbolo de entrada del alfabeto de entrada coincide con los estallidos simbólicos de la pila, también elimina el punto de la esa es la entrada que se el punto se mueve un paso adelante si esto no es así, el reflejo no es igual a , eso significa que la pila y la entrada no lo Entonces, se debe informar un error. La siguiente posibilidad es que la parte superior de Stack sea un símbolo no terminal, aquí hay un símbolo terminal ahora, es un T no terminal.

M es la tabla de personas si la coma N, M, X a entonces X es el no terminal que se expandirá y a es el símbolo de entrada la siguiente entrada símbolo.

Entonces, si esta es una combinación que le da una producción única y única x_1 en Y en $y_1 Y_2$ y k , entonces sabemos que es hora de abrir la pila y luego expandir el símbolo X bytes a la derecha. Entonces el lado derecho en el orden inverso por K por K menos 1 Etc por uno si yo uno en la parte superior es empujado a la pila. Así que ahora uno está encima de entonces y volvemos a repetir hasta que Loop elimine el siguiente símbolo de pila de parada.

Entonces estas son las filas indexadas por los no terminales y las columnas indexadas por los símbolos de terminal son el símbolo del final del archivo, por lo que para cada una de estas combinaciones S Prime y S Prime y v s Prime y C S Prime y dólar exactamente una así que algunas de las entidades también pueden estar vacías. Por ejemplo, S Prime y V está vacío S Prime y el



dólar es el enésimo término de s_n vacío y también vemos que ninguno de los espacios tiene más de un mentor.

No reciben ningún error, luego se pone también vacío. Entonces la cadena ha sido aceptada con así que así es como el algoritmo de análisis de LL 1 pasa repetidamente por el empuje de las etapas de expansión de Pop en el algoritmo. Entonces ahora es el momento de entender cómo se construye exactamente la tabla de personas. Entonces, como dije, las gramáticas LL 1 son una subclase de gramáticas libres de contexto. Así que cuando decimos Excedente debemos poner restricciones a la gramática libre de contexto nuevamente, tu método para verificar si las restricciones se cumplen o así que definamos una clase de gramáticas llamadas gramáticas LL fuertes

-video 2-

Condiciones comprobables para $II(1)$

Puede ser un símbolo de terminal o el final de la tabla de símbolos de Entonces lo que tenemos que hacer es entender es que en primer lugar, por qué está determinado solo por cómo sigue

Se sabe muy simplemente con First Step como primero que no es más que el primer que son todas las cadenas de S Prime en realidad derivadas de S Y que si derivan todas las cuerdas que son, ya sabes derivadas por S comienzan con a o C . Así que primero de s sería una C

Literalmente por el primero de II y Tienes todas las cadenas derivables de un sabes que comienzan con la pequeña B . Y porque hay una S no termina aquí todas las cadenas que son derivables por S también están incluidas en la primera forma. y para B , entonces eso nos da a saber, el primero de a tiene $a b c$ en este momento en lo que respecta a B , los símbolos son y el primero de los suyos están incluidos en el primero, así es como se completa todo esto.

Seguir de aquí de manera similar es una C porque S Prime Darrell, CA. Lo que respecta al ejemplo para calcular los primeros cuatro terminales y no Son símbolos de terminal bastante sencillos, el primer set no gana, es el símbolo en Y el primero de Epsilon siempre está en silencio. Para que uno de terminal comenzamos con cinco y luego consideramos y necesitamos valorar el cálculo, todo esto hasta que incluso uno que cada primer conjunto deje de cambiar. Así que incluso si uno de ellos está cambiando, tenemos que intentarlo una vez más, esto se debe a que la competencia del primer set (conjunto), incluso si uno de ellos cambia, en realidad puede hacer que otros cambien también.

Entonces el cálculo es bastante complejo, la primera señal de producción vamos a $x_1 \times 2$ etc x_n ahora primero a obviamente incluye el primero de x_1 . Así que primero somos iguales a Somos la Unión primero de x_1 menos Epsilon. Ahora si exiliamos x_1 en realidad primero de x_1 productores en silencio en ese caso, lo que sea deseado por x_2 también se divide por a , por lo que eso es lo que se marca aquí es Epsilon en pasta KET PSI.

Así que si es así, incluimos el primero de x_2 también en la primera mitad aquí de manera similar si hay X_3 y x_1 y x_2 ambos producen Epsilon, entonces codificamos, el primero de x_3 también para Así que esto continúa y si $x_1 \times 2$ etc x_n todos ellos una especie de en primer lugar, estos símbolos contienen Epsilon. Entonces esos son todos los símbolos que se han visto, eso significa que todos ellos han reducido Epsilon y Epsilon está en el primero de x_n . Así que ese es el último al que vamos a Epsilon a la primera configuración.

Así que supongamos que tenemos una cadena beta para R que queremos calcular el primer conjunto que no es diferente de calcular el primer conjunto de un no terminal. si la cadena está



sentada, beta es de x_1 a X_N , hacemos exactamente lo que hicimos antes para los no terminales, excepto que este algoritmo ha colocado beta en lugar de VA en todos los Así que consideramos $x_1 \times 2$ etc x Entonces, el primero de x_1 está obviamente en posoperatoria

al principio inicializamos los primeros conjuntos de todos los terminales notn para luchar contra el conjunto nulo y la iteración uno. Primero está la celda Z Comet porque deriva ay también va a Epsilon. Así que eso incluye una excelente primera serie.

CA Epsilon es la primera configuración. Entonces esto es en lo que respecta a la primera iteración en "n" hasta ahora la primera vez que ha ayudado a vender el tiene las cosas que tenemos.

Así que eso significa que esto se convierte en V. AC de es diferente. Primero de b c Epsilon no tiene cambio no ha cambiado desde el último elemento. Entonces estos son los valores que se estabilizaron y no cambian de forma aislada.

En un conjunto de cada "n" Terminal 2 cinco s es el símbolo de inicio, entonces tenemos S Prime yendo a S D por D siempre estará presente. Así que hacemos esto para cada produccion. Hasta ahora sigue que ha cambiado exactamente la forma en que calculamos este primer conjunto.

Entonces en otras palabras. Cualquier símbolo que Alpha derive y beta desarrolle los primeros símbolos de estos no deberían ser los De lo contrario, la elección no se puede hacer mirando el siguiente símbolo de entrada. Entonces hay una formulación equivalente en el libro cómo se conocen. Dice en primer lugar que el DOT siga una primero del punto beta sigue un igual a 5 para los mismos productos y C 2 Alfa y beta beta estas condiciones son Es fácil ver eso porque supongamos que Alpha no produce ningún Epsilon en ese caso Epsilon inverso de alfa es falso. Entonces el conjunto de símbolos de dirección simplemente se convierte en primer lugar para, por lo que lo siguiente no tiene consecuencias en ese caso.

Supongamos que Alpha produce Epsilon. Así que en primer lugar, los contiene aplíquelos en tal caso si este es nuestro Epsilon, entonces el siguiente conjunto de a también se incluirá en la operación de la primera Así que también hacemos eso aquí, ya sabes si Epsilon está en el primero de los estallidos gamma de gamma menos X_{Ln} Unión durante de a así que todos los elementos en el seguimiento de a también se incluirán en el espejo, todo Encontraré que este caso también es cierto para este. Primero del punto beta siga aquí. Entonces, la intersección del símbolo de dirección es esta intersección no idéntica en la medida en que se cumple la condición L1


Entonces esta es la condición para que el apisonador satisfaga la propiedad de análisis sintáctico LL1. Supongamos que se cumple la condición. ¿Podemos construir una tabla de análisis sintáctico a partir de la Definitivamente el proceso es bastante simple para cada producción a Alpha que destruye cada símbolo S en el símbolo de dirección Alpha. Tal vez un símbolo de terminal o el final de un dólar considerable, solo agregue la producción pero la pared para lo positivo en el punto una coma s, esto y haremos cada entrada indefinida como un error. Entonces con la otra formulación primero y siga la La construcción de la mesa es muy similar, considere la primera de todas las reducciones profundas, y si Epsilon es inverso de alfa, agregue para todas las caídas a también y si el D está en el seguimiento, se agregan también para el D. Entonces, después de la construcción de la tabla, por supuesto, podríamos probar las predicciones para la propiedad LL 1 y luego el que podríamos construir una tabla y verificar si alguna ranura en la tabla L1 tiene más de un proyecto para ser la gramática no es, entonces estas dos condiciones son idénticas.



Eliminación de recursión izquierda

Elimination of Left Recursion

- A *left-recursive* grammar has a non-terminal A such that $A \Rightarrow^+ A\alpha$
- Top-down parsing methods (LL(1) and RD) cannot handle left-recursive grammars
- Left-recursion in grammars can be eliminated by transformations
- A simpler case is that of grammars with *immediate left recursion*, where there is a production of the form $A \rightarrow A\alpha$
 - Two productions $A \rightarrow A\alpha_1 \mid \beta$ can be transformed to $A \rightarrow \beta A', A' \rightarrow \alpha_1 A' \mid \epsilon$
 - In general, a group of productions:
 $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
can be transformed to
 $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A', A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$



YN Srikant
Parsing

El problema es que las gramáticas recursivas izquierdas no satisfacen la condición LL 1.

Entonces, si una gramática tiene un no terminal A tal que A en más de una aplicación de las producciones de A produce A alfa.

El primer símbolo de esta forma de oración también es A , lo que significa que puede producir tantas, sabes que podemos ir y aplicar las producciones de A tantas veces como queramos y esto se conoce como recursividad por la izquierda.

Entonces, nuestro método de análisis de LL 1 y también el uno, veremos que el método de análisis de descenso recursivo no puede manejar el recursivo izquierdo gramáticas. Por tanto, las gramáticas de recursividad por la izquierda pueden, por supuesto, eliminarse, en otras palabras puede convertir la recursividad izquierda en recursión derecha y, de forma similar, la recursión derecha se puede convertir a la recursividad izquierda.

Entonces, en primer lugar, observando lo que se conoce como recursión izquierda inmediata, entonces necesitamos manejar producciones de la forma A que van a un alfa, recuerde que la recursividad a la izquierda implica que A deriva A alfa en una o más aplicaciones de producciones de A .



Considerando que, aquí la producción en sí tiene no terminal A a la izquierda más en la posición más a la izquierda. Entonces, si hay una producción de la forma A que va a A alfa entonces la gramática está configurada para tener recursión izquierda inmediata.

Hay dos producciones A yendo a A alfa y A yendo a beta, el supuesto es que A yendo a beta no tendrá A en la posición más a la izquierda.

Entonces, si es así, entonces podemos transformar estas dos producciones en estas tres producciones.

En primer lugar, A va a beta A prima y luego A prima va a alfa A prima o A prima yendo a epsilon, A prima es un nuevo no terminal, que no existía antes. Entonces, sabemos que esta aplicación produce exactamente beta A prima y nada más, pero entonces un primo puede ser que sepa que tiene recursividad correcta. Entonces, podemos usarlo tantas veces como queramos. Entonces, A prima que va a alfa A prima se puede usar muchas veces para producir muchas instancias de A.

Observando esta producción, producimos tantas instancias de alfa como quisiéramos y luego A fue reemplazado por beta. Entonces, aquí también A prima produce tantas instancias de alfa como queramos. Y finalmente, A prima va a epsilon allí al no producir nada.

Left Recursion Elimination - An Example

$$A \rightarrow A\alpha \mid \beta \Rightarrow A \rightarrow \beta A', A' \rightarrow \alpha A' \mid \epsilon$$

- The following grammar for regular expressions is ambiguous:

$$E \rightarrow E + E \mid E E \mid E^* \mid (E) \mid a \mid b$$

- Equivalent left-recursive but unambiguous grammar is:

$$E \rightarrow E + T \mid T, T \rightarrow T F \mid F, F \rightarrow F^* \mid P, P \rightarrow (E) \mid a \mid b$$

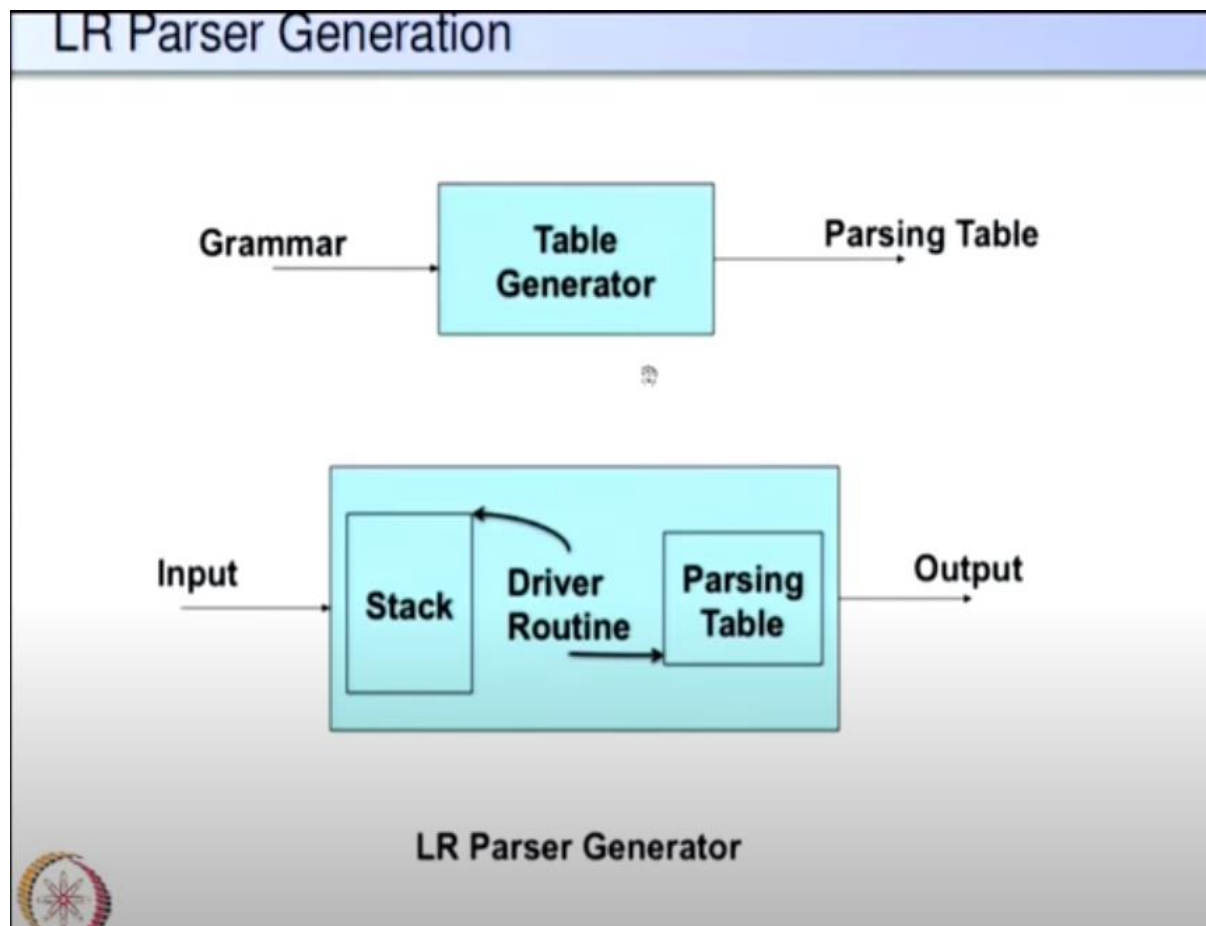
- Equivalent non-left-recursive grammar is:

$$E \rightarrow T E', E' \rightarrow + T E' \mid \epsilon, T \rightarrow F T', T' \rightarrow F T' \mid \epsilon, \\ F \rightarrow P F', F' \rightarrow * F' \mid \epsilon, P \rightarrow (E) \mid a \mid b$$



Pero ya hemos producido una versión beta para empezar. Hemos producido beta seguido de muchos alfas, que es exactamente lo mismo que A producido. Entonces, esta es la recursión izquierda inmediata.

Generador de analizador sintáctico



Este es el diagrama de bloques de un generador de analizador sintáctico LR. Entonces, la gramática se ingresa en una tabla generador este es el generador de tablas de análisis sintáctico LR y da como resultado una tabla de análisis sintáctico. Y cuando nosotros utilicemos la tabla de análisis, también hay una rutina de derivación, esto es, este bloque está completo analizador. Entonces, el analizador consta de una pila, una rutina de derivación y una tabla de análisis que se generan automáticamente.

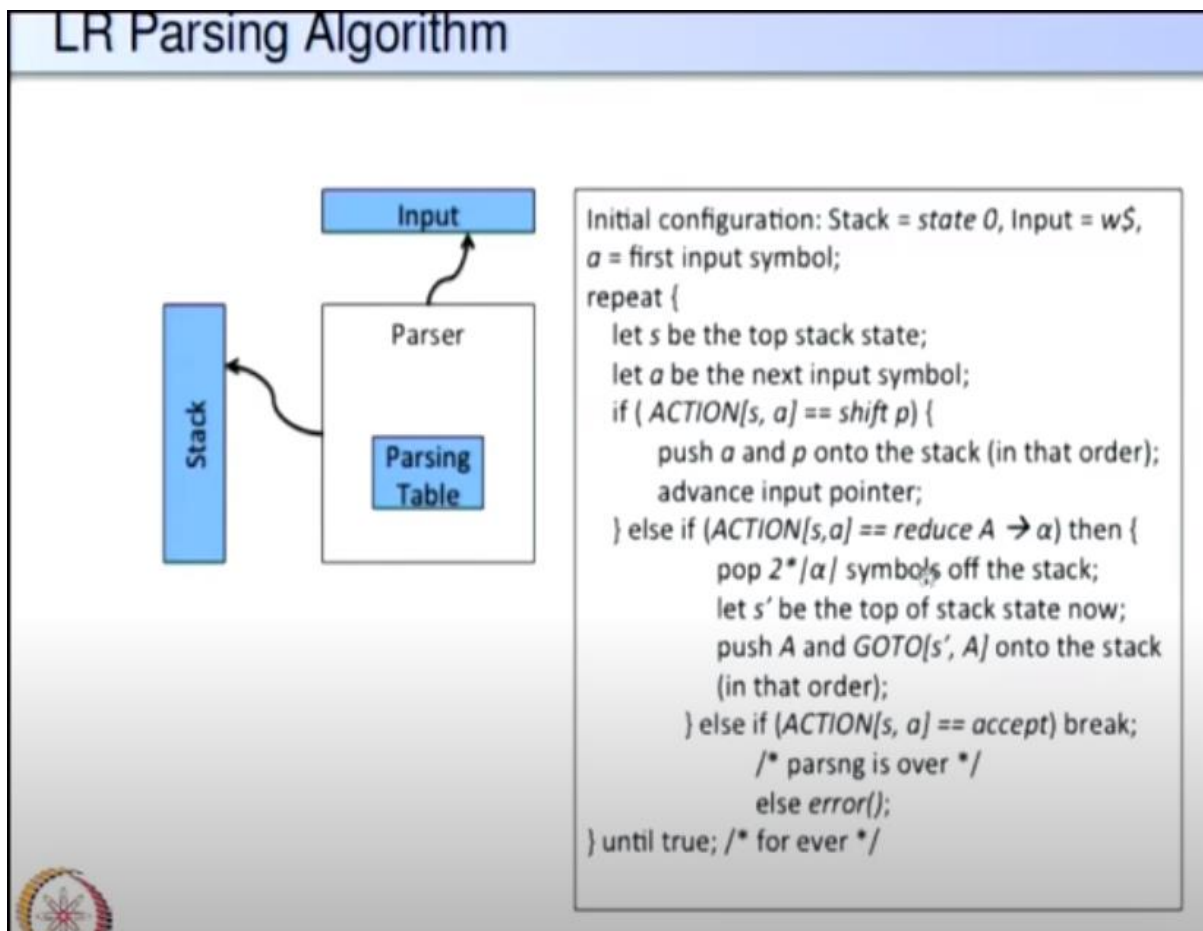
Entonces, cuando se da la entrada, la rutina de derivación lee la entrada manipula la pila apropiadamente usando la tabla de análisis y los generadores algunos salida en forma de árbol analizador o mensajes de error, etc.

La tabla de acciones puede tener cuatro tipos de entradas: shift, reduce, accept y error.

- **Shift:** el siguiente símbolo de entrada se desplaza hasta el tope de la pila.

- **Reduce:** el extremo derecho del handle es la parte superior de la pila; ubica el extremo izquierdo del mango dentro de la pila y reemplaza el mango por el LHS de una producción adecuada
- **Accept:** anuncia la finalización exitosa del análisis.
- **Error:** Error de sintaxis, se llama a la rutina de recuperación de errores.

Algoritmo de análisis sintáctico LR.



Entonces, aquí está el analizador tiene una tabla, tiene una pila y también mira la entrada. Comienza con la inicial estado 0 y la entrada es w dólar, a es el primer símbolo de entrada. Entonces, todo es repetido para siempre hasta que salgamos de él.

Sea S el estado superior de la pila, a sea el siguiente símbolo de entrada. Entonces, miramos S y a en la parte de acción, si es un cambio P, presione ay P en la pila en ese orden, avance el puntero de entrada.

Entonces, si la acción dice reducido de A a alfa, saque dos símbolos alfa de estrella de la pila la razón es que la pila tiene una combinación de estados y símbolos gramaticales. Por lo tanto, allí son dos símbolos m, si estamos al lado derecho del, si el mango es del tamaño n.



Realmente hacemos estallar dos símbolos alfa de estrella de la pila. Ahora, el estado S prima está expuesto en la pila. Entonces, presione a y goto de S prima con a en la pila en ese orden.

Entonces, así es como determinamos el siguiente estado al que se va a saltar. Si la acción es aceptar luego terminamos de otra manera, anunciamos un error y salimos, ya sea para hacer una recuperación de error o salir del analizador.

Actividades semana 7 (Nov 2-6, 2020) Tarea

Un autómata finito tiene un conjunto de estados y su "control" pasa de un estado a otro en respuesta a las "entradas" externas. Una de las diferencias fundamentales entre las clases de autómatas finitos es si dicho control es "determinista", lo que quiere decir que el autómata no puede encontrarse en más de un estado a un mismo tiempo, o "no determinista", lo que significa que sí puede estar en varios estados a la vez. Comprobaremos que añadir el no determinismo no nos permite definir ningún lenguaje que no pueda ser definido mediante un autómata finito determinista, aunque se obtiene una eficacia sustancial al describir una aplicación utilizando un autómata no determinista. En efecto, el no determinismo nos permite "programar" soluciones para los problemas utilizando un lenguaje de alto nivel.

análisis y análisis de arriba hacia abajo recursivo análisis de descenso y un poco de abajo hacia arriba analizando así que continuemos con LR analizando técnicas hoy para que yo explicado en la conferencia anterior LR el análisis sintáctico es un método de análisis sintáctico ascendente y significa escaneo de izquierda a derecha con derivación más a la derecha en Reverse y K es el número de tokens de cabeza locales de Por supuesto, LR 0 y LR 1 son de gran interés para nosotros en el sentido práctico LR los analizadores, por supuesto, son importantes porque se pueden generar automáticamente usando generadores de analizador y son un subconjunto de gramáticas LR libres de contexto son un subconjunto de gramáticas libres de contexto para que se pueden construir tales analizadores es fácil escribir gita nuestras gramáticas y esa es la razón por la que son muy popular hoy, así que veamos el generador de analizador sintáctico el generador es muy dispositivo simple que toma la gramática como entrada y genera una tabla de análisis denominada Tabla de análisis sintáctico LR y la tabla analizador es encajar en otra caja que contenga una pila y una rutina de conductor, esto es todo la configuración es el paquete, así que aquí, por ejemplo así que la rutina del conductor del ciervo y la tabla de análisis juntos hacemos el analizador toma el programa como entrada y entrega como salida posiblemente una sintaxis árbol o algo más, así que veamos los operación parsa para entenderlo

Pasador LR entonces la configuración tiene dos partes una es la pila el otro es el no gastado o entrada no utilizada y para comenzar con el pila tiene sólo el símbolo de inicio o el estado inicial del analizador y la entrada no gastada tiene toda la entrada

paréntesis ID paréntesis hash y nosotros obtener una forma de oración correcta, así que esto es la característica de un prefijo viable prefijos viables caracterizan el prefijos de forma oracional que pueden ocurren en la pila de un analizador LR por lo que cuando pasamos del símbolo terminal cadena terminal al símbolo de inicio a se llevan a cabo muchas reducciones y durante estas reducciones la pila contiene partes de las formas oracionales y Los prefijos viables son exactamente aquellos centros que conoces partes que se encuentran en el pila de un rompecabezas de lr un teorema principal en La teoría del análisis sintáctico LR es que por un conjunto de prefijos viables de todos los derechos enviados formas orales de una gramática el conjunto



forma un lenguaje regular para que el DFA de este idioma puede detectar identificadores durante Análisis LR, así que veremos que muy pronto el punto es que el DFA alcanza un los llamados estados y señales de reducción que el prefijo prefijo viable no puede crecer más, lo que significa que hay un reducción que es necesaria en este punto te mostraré qué es exactamente un estado de reducción después de esta diapositiva este una especie de DFA puede ser construido por el compilador usando la gramática y estamos voy a discutir ese procedimiento un poco más tarde todos los gritos o personas han tal DFA incorporado en ellos esto es el corazón de y muchos de los nuestros son realmente buenos que construimos una gramática aumentada y si ves el símbolo de inicio de G entonces G prime contiene todos los producciones de G junto con el nuevo la producción s prime va a sí el la razón por la que hacemos eso es que podría haber producciones de yes con YES en el lado derecho también, pero queremos asegúrese de que el símbolo de inicio esté único y queremos corazón el analizador

Para hacer un poco de resumen. Analizamos el concepto de elementos de prefijo viables elementos válidos. Y entonces también discutimos el procedimiento para la construcción del autómata LR(0), mostrado la última vez.

Y el método de construcción es muy simple, usted sabe que para comenzar con el elemento S primo para el punto S es el único que vamos a y luego tomamos el cierre en él. Para le da un conjunto inicial de elementos y ahora aplicamos continuamente la función ir a la función y seguir incluyendo tantos conjuntos de artículos como sea posible en la colección C. Y cuando no podemos añadir más artículos que detenemos. Por lo tanto, cada conjunto de la colección anterior es un estado de la LR(0) DFA, y este DFA reconoce los prefijos viables. Por lo tanto, cada estado del DFA contiene sólo elementos que son válidos para un prefijo viable en particular o un conjunto de prefijos viables.

A Grammar that is not SLR(1) - Example 2

<p>Grammar</p> <p>$S' \rightarrow S$</p> <p>$S \rightarrow L=R$</p> <p>$S \rightarrow R$</p> <p>$L \rightarrow *R$</p> <p>$L \rightarrow id$ ⁽²⁾</p> <p>$R \rightarrow L$</p>	<p>State 0</p> <p>$S' \rightarrow .S$</p> <p>$S \rightarrow .L=R$</p> <p>$S \rightarrow .R$</p> <p>$L \rightarrow .*R$</p> <p>$L \rightarrow .id$</p> <p>$R \rightarrow .L$</p>	<p>State 2</p> <p>$S \rightarrow L.=R$</p> <p>$R \rightarrow L.$</p> <p>shift-reduce conflict</p>	<p>State 6</p> <p>$S \rightarrow L=.R$</p> <p>$R \rightarrow .L$</p> <p>$L \rightarrow .*R$</p> <p>$L \rightarrow .id$</p>
	<p>State 1</p> <p>$S' \rightarrow S.$</p>	<p>State 4</p> <p>$L \rightarrow *.R$</p> <p>$R \rightarrow .L$</p> <p>$L \rightarrow .*R$</p> <p>$L \rightarrow .id$</p>	<p>State 7</p> <p>$L \rightarrow *R.$</p>
	<p>State 3</p> <p>$S \rightarrow R.$</p>	<p>State 5</p> <p>$L \rightarrow id.$</p>	<p>State 8</p> <p>$R \rightarrow L.$</p>
		<p>State 9</p> <p>$S \rightarrow L=R.$</p>	

Grammar is neither LR(0) nor SLR(1)

Follow(R) = { \$, = } does not resolve S-R conflict



Entonces, entendamos por qué ocurrió esta falla en particular en el último ejemplo o en el anterior, el problema es que los analizadores sintácticos S L R 1 consideran que solo usted conoce todos los símbolos a continuación.

El proceso de construcción no recuerda suficiente contexto izquierdo para resolver los conflictos. Entonces, tomemos la gramática I igual a R que es esta gramática en particular, veamos cómo el símbolo igual entró en el siguiente conjunto.

Entonces, comenzamos con S prima deriva S y luego deriva I igual a R. Ahora, esto deriva I igual a I que deriva I igual a i d esta es la derivación más a la derecha. Por lo tanto, esto debe derivarse primero y luego se aplica I a la estrella R y obtenemos la estrella R igual a i d. Entonces, ahora, igual a lo que sabes está en el siguiente conjunto de R que es fácil de ver, porque esta es una forma enunciada, el problema es.

Usted sabe que no es posible reducir el R solo a I, si es que lo reducimos, debe ser la estrella R reducir a I y no es posible tener ninguna reducción de R a I. Porque sabes que realmente no tenemos una producción de ese tipo, la única producción que tenemos es la estrella R que se reduce a I, ir a la estrella R es la única producción, mientras que en este caso dice que sabes, que I se reduzca a R correcto.

Entonces, la reducción de 1 a R o de R a 1 nunca puede ocurrir en forma de oración. Entonces, esta es la dificultad básica, la siguiente derivación inversa del extremo derecho no existe en absoluto, hay una reducción de la R a la derecha.

Es decir, digamos R igual a i d, lo que nos da I igual a i d, lo que nos da i d igual a I d. Entonces, si tuviéramos esto, habríamos reducido I 2 r. Entonces, de manera similar, la reducción de R a 1 por sí sola no puede ocurrir. Entonces, en igual a la moraleja de la historia es igual a la reducción debería haber sido la estrella R a I. Y nunca es correcto reducir lo que sabes de 1 a R solo porque sabes que aquí está la reducción de 1 a R, pero no fue por el símbolo igual a, sino por una razón diferente. Por lo tanto, nunca obtendremos una forma enunciativa y una derivación en la que podamos reducir 1 a R viendo en igual que eso sería correcto. Entonces, esta es la razón por la cual la estrategia S L R 1 falló en el ejemplo anterior.

Entonces, si tratamos de generalizar lo anterior, sea lo que sea lo que dije aquí, el punto es en algunas situaciones cuando un estado I aparece en la parte superior de la pila.

ANÁLISIS SINTÁCTICO PARTE 7

El analizador sintáctico ascendente LR(1) es el analizador más costoso de implementar, aunque es el que reconoce más gramática. El método para construir los conjuntos de elementos es básicamente el mismo que el utilizado en el SLR, aunque incorpora el símbolo de lookahead o símbolo de anticipación a los elementos del conjunto.

CONSTRUCCION DE LA TABLA DE ANÁLISIS

Una vez hemos obtenido el AFD LR(1) se construye la tabla de análisis sintáctico LR(1), a partir de los estados obtenidos y donde se obtendrán las acciones a realizar para las entradas de la matriz M[estado, terminal o no terminal], donde M podrá ser la parte acción si los



símbolos son terminales o \$, o bien la de ir_a si se trata de un no terminal. El algoritmo de construcción consiste en realizar las acciones siguientes para cada uno de los estados.

Conflicto de reducción/desplazamiento	Se produce cuando en un mismo estado existe una producción con un elemento completo, del tipo $[A \rightarrow \alpha., a]$, y otra producción del tipo $[A \rightarrow \alpha.T\beta, a]$, donde T es alguno de los símbolos de anticipación (a_1, a_2, \dots, a_n) de la producción $[A \rightarrow \alpha. a]$.
Conflicto de reducción/reducción	Se produce cuando en un mismo estado existen dos producciones con el elemento completo, del tipo $[A \rightarrow \alpha., a]$ y $[B \rightarrow \beta., a]$, para el mismo símbolo de anticipación (a), no pudiendo por tanto decidirse qué reducción aplicar

CONSTRUCCIÓN DEL ANALIZADOR LALR(1)

Al realizar análisis sintáctico SLR y LR(1), es importante saber que los LR(1) trabajan con gramáticas que son ambiguas para los SLR, aunque para ello necesitan generar autómatas con más estados. Esto se debe a la utilización de los símbolos de anticipación.

Con los analizadores sintácticos LALR se trata de conseguir los beneficios que proporciona el método LR(1), pero con el coste del método SLR y utilizando, por tanto, el menor número de estados que este último necesita. Es decir, este método consigue reducir el número de estados.

CARACTERÍSTICAS

LALR: Look Ahead Left to Right Analisis sintactico con simbolo de anticipacion.

Estrategia Se basa en la unificacion de conjuntos de elementos-LR(1) que tienen los mismos centros.

Potencia Mas potente que el metodo de analisis SLR Menos potente que el metodo de analisis LR-canónico
Tamaño de la tabla La tabla LALR tiene el mismo tamaño que la tabla SLR.



PROCESO

PRIMER PASO	Se seleccionan en el autómata LR(1) los estados con el mismo núcleo y se crea uno nuevo que es la unión de estos. El núcleo del nuevo estado será el núcleo común y como símbolo de anticipación tendremos la unión de todos los símbolos de anticipación de cada uno de los estados del conjunto.
SEGUNDO PASO	Para cada estado del autómata modificado, hacemos lo siguiente: Si es un estado no modificado (para cada arista que salga de él). Si el estado de llegada es un estado modificado, (unión de otros o ha desaparecido) entonces la arista irá al estado unión correspondiente al modificado. Si es un estado modificado. Todas las aristas que parten de este estado tienen que partir de la unión correspondiente y si el estado al que llegaban estas aristas es modificado entonces llegarán a la unión correspondiente.

Estados con núcleo común

Estados 2 y 6 (2_6): S ((. L) y como símbolo de anticipación {\$} U {,, }), quedando \$ |, |).

Estados 3 y 7 (3_7): S (id. y como símbolo de anticipación {\$} U {,, }), quedando \$ |, |).

Estados 4 y 11 (4_11): S ((L.) y como símbolo de anticipación {\$} U {,, }), quedando \$ |, |).

Estados 8 y 13 (8_13): S ((L). y como símbolo de anticipación {\$} U {,, }), quedando \$ |, |).

CONSTRUCCION DE UNA TABLA DE ANALISIS

YACC

Es un programa para generar analizadores sintácticos. Las siglas del nombre significan Yet Another Compiler-Compiler, es decir, "Otro generador de compiladores más". Genera un analizador sintáctico (la parte de un compilador que comprueba que la estructura del código fuente se ajusta a la especificación sintáctica del lenguaje) basado en una gramática analítica escrita en una notación similar a la BNF. Yacc genera el código para el analizador sintáctico en el Lenguaje de programación C.

ANALIZADOR LEXICO

1. Para realizar la función del analizador de léxico, el Yacc utiliza una función entera llamada `yylex()`, cuyo valor se debe asociar a la variable `yyval`
2. El analizador de léxico:



3. puede ser la función generada con LEX, o
4. una función definida por el usuario en la zona de rutinas de usuario
5. ESTRUCTURA DE ESPECIFICACION YACC

{ Declaraciones de los símbolos }

%%

Reglas gramaticales {+acciones}

%%

{ Rutinas de usuario }

Se pueden incluir comentarios con el siguiente formato:

/* cero o más caracteres */

TRATAMIENTO DE ERRORES

1.- Tratamiento por defecto: en una situación de error se produce una llamada a la función yyerror con el mensaje "Syntax Error" y finaliza la ejecución del analizador sintáctico

2. Introducción de producciones de error: en la especificación YACC se introducen producciones denominadas de error que tienen el siguiente formato:

A ® error w

donde A es un símbolo no terminal de la gramática, error una palabra reservada de YACC y w es un símbolo terminal.

El YACC generará una tabla de análisis a partir de la especificación dando a las producciones de error el mismo tratamiento que al resto de las producciones

Actuación del analizador en una situación de error:

Sea S el estado del tope de la pila:

Si para S existe un desplazamiento con el símbolo error lo aplicará

Si para S no existe un desplazamiento con el símbolo error, extrae símbolos de la pla hasta que encuentra un estado que incluya un ítem de la forma A error w introduce error en la pila

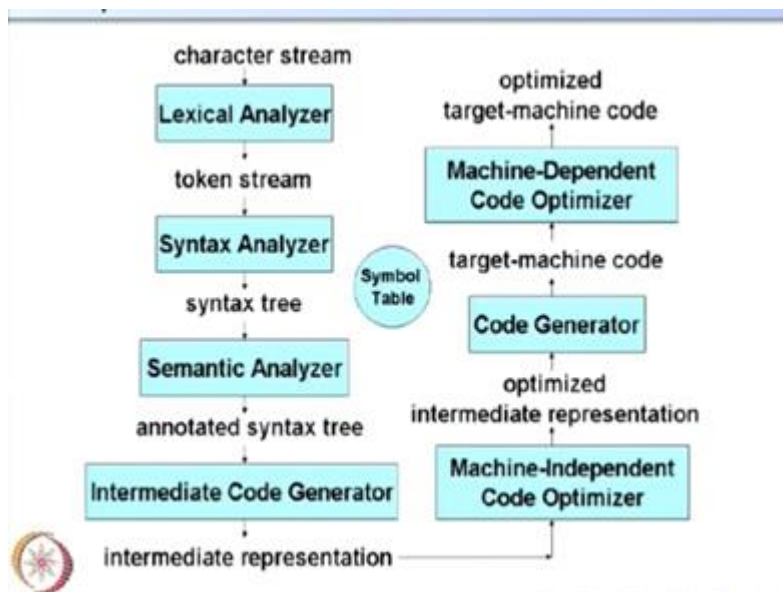
Si w es distinto de la cadena vacía, eliminará caracteres de la entrada hasta encontrar w. Entonces pasará a reducir por la producción de error.

Si w es la cadena vacía pasa a reducir por la producción de error

Actividades semana 8 (Nov 9-13, 2020) Tarea

Las gramáticas de atributos

atribuidas a las gramáticas de traducción y cómo El análisis semántico se realiza con estas gramáticas de traducción atribuidas estas son los temas principales y vemos los subtemas también para poner el módulo de análisis semántico en la perspectiva correcta una descripción general una vez más, así que hemos completado la sintaxis del analizador léxico



En el Analizador y estudió sus propiedades diferentes tipos de tales analizadores y así de modo que desde el analizador de sintaxis, el árbol de sintaxis se ingrese en la semántica módulo analizador que genera este árbol de sintaxis anotado después de validar la semántica del programa, entendemos qué es exactamente la semántica de un programa significa cómo se comprueban, así que básicamente semántica.

La consistencia que no se puede manejar en la etapa de análisis es en realidad la única que se maneja aquí, por ejemplo, los analizadores no pueden manejar características sensibles al contexto de lenguajes de programación, por lo que hay dos tipos de semántica que la programación



Static Semantics

```
int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}
main(){
    int p; int a[10], b[10];
    p = dot_prod(a,b);
}
```

Samples of static semantic checks in *main*

- Types of *p* and return type of *dot_prod* match
- Number and type of the parameters of *dot_prod* are the same in both its declaration and use
- *p* is declared before use, same for *a* and *b*

los idiomas tienen uno se conoce como semántica estática y el otro se conoce como la semántica dinámica la semántica estática como el nombre indica que sabe no dependen del sistema de ejecución y de la ejecución del programa, sino dependen solo de la definición del lenguaje de programación, mientras que la semántica dinámica de nuevo, ya que el nombre lo indica, son las propiedades de la sistemas de lenguaje de programación Tucker en tiempo de ejecución y tenemos que comprobar tales propiedades solo durante el tiempo de ejecución del programa, por ejemplo, la semántica estática puede ser verificada por eso puede ser verificada por un analizador semántico no hay muchos ejemplos aquí, por lo que nuestras variables declaradas antes de su uso si es así, todo está bien, de lo contrario, se debe proporcionar un mensaje de error, entonces los tipos de expresión y la variable a la que está asignada coinciden en dos lados de una declaración de asignación y hacer los tipos de parámetros y el número

Semántica estática de lenguajes de programación donde, como sabe, para dinámica los compiladores de semántica solo pueden generar código para verificar el significado que conoces del programa, por ejemplo, si se producirá un desbordamiento durante un 20

textos semánticos estáticos en el producto escalar de la función, así que vamos a saber considerar algunas de las dinámicas controles semánticos en el producto escalar para que el valor de *l* no exceda el rango declarado de borrar *x* y tanto inferior como superior, por lo que *x* y en el programa principal se han declarado con talla 10 y por alguna razón si decimos que es talla 11 o 12 *l* menos de 11 o menos de 12, entonces sabes que estaríamos accediendo a *X* 11 e *Y* 11 *X* 12 e *Y* 12 etcétera que no están dentro del rango de la matriz declaración, por lo que habría un error de tiempo de ejecución en este punto, no hay en este caso, por supuesto, *X* sabes que voy de 0 a 9 y eso son solo 10 iteraciones, así que todo está bien dentro de la declaración de preguntas que sabe del para borrar *X* & *Y* no hay toda la fuerza durante las operaciones de estrella y más en *D* es igual a *D* más *X* *l* estrella *Y* *l* nuevamente los enteros que somos si los valores de las variables *C* y *B* las matrices de componentes de *a* y *B* resultan ser valores extremadamente grandes, entonces la multiplicación *X* *l* estrella por una podría en realidad, provocará que se produzca un desbordamiento de manera similar nuevamente si estos valores, si son si las matrices son demasiado grandes y se están multiplicando demasiados componentes y agregado en *D* luego *D* podría nuevamente, usted sabe cruzar el límite y causar un desbordamiento pero en este caso, suponiendo que *n* tenemos valores pequeños, no ocurren, por lo que estos son algunos ejemplos de lo que exactamente controles semánticos dinámicos en programas que pueden ser que no pueden ser realizados por el compilador, pero un compilador definitivamente



puede generar código para realizarlos, lo que sucede es comprobar que la matriz ya sabe que se sale de los límites en tiempo de ejecución, se comprueba que el valor del índice del sea menor que 10 si es entonces el se permite que la iteración vaya más allá, de lo contrario se producirá un error y el programa se detiene bien, por lo que el otro tipo de verificación de semántica dinámica que puede tomar

Lugo se demuestra de nuevo con el uso de una función recursiva de hecho, por lo que este calcula el factorial de un número, por lo que si n es igual a 0, devuelve 1; de lo contrario, devuelve n hecho de n menos 1 este es un programa muy conocido, por lo que no es necesario explicarlo más, de modo que P es un número entero y P es igual al hecho n , por lo que estamos tratando de calcular 10 factorial, la pila del programa se desborda debido a recursividad así que aquí cuando decimos un hecho de 10, aquí viene 10 en un hecho de 9, luego de nuevo 9 en hecho de 8, etc, por lo que en realidad hay 10 invocaciones de hecho

Análisis semántico por lo que durante el análisis semántico el tipo información de las variables, ya sean ins o matrices o pullian o caracteres, etc. se almacena en lo que se conoce como la tabla de símbolos o la sintaxis árbol para que pueda almacenarse en la tabla de símbolos o podría almacenarse en los nodos del árbol de sintaxis en sí, por lo general, la opción de tabla de símbolos es mejor

El árbol de sintaxis ocupa mucho más espacio que la tabla de símbolos en la que normalmente lo almacenamos una tabla de símbolos y luego coloque un puntero a la entrada correspondiente en el símbolo tabla en uno de los campos en el nodo del árbol de sintaxis, el nombre de la variable, etc, se almacenan en el símbolo tabla, por lo que estas tablas de símbolos se utilizan no solo para la validación semántica del análisis semántico,

semántica estática de los lenguajes de programación y cómo especificarlos usando los índices semánticos de las gramáticas de atributos se pueden generar semiautomáticamente a partir de gramáticas de atributos para que escribamos la semántica para la que conoce las especificaciones el programa para el lenguaje de programación y luego podríamos usar un generador que los genera a partir de tal atributo gramatical la razón por la que decimos semiautomáticamente es que la semántica generalmente se especifica en una programación lenguaje como la notación, no es una notación pura libre de efectos secundarios, así que esa es la razón por la que se llama semiautomático, el programador tiene que secuenciar el verificaciones semánticas apropiadamente y luego enviarlo al generador de hecho también es un generador de analizadores semánticos si proporcionamos las apuestas semánticas de manera apropiada en las gramáticas de atributos C son extensiones de gramáticas libres de contexto

Gramáticas con cierto detalle sucede que las gramáticas de atributos requieren que se comprenda alguna terminología

Dichas gramáticas de atributos no son más que extensiones de gramáticas libres de contexto, por lo que la base es definitivamente una gramática libre de contexto y deja que el conjunto de variables V de la gramática estar en unión P para cada símbolo de X de V podemos asociar un conjunto de atributos denotados como X punto e ax punto B etc.

Por eso la nombre gramática del atributo hay dos tipos de atributos heredados que se denotan como una L de X , por lo que heredaron atributos de X y Atributos sintetizados que se denotan por X , estos son realmente conjuntos de atributos cada atributo toma valores de un dominio específico podría ser un dominio finito o podría ser un dominio infinito y llamamos al dominio



Un dominio como su tipo, por ejemplo, incluso en lenguajes de programación decimos `int` de X , por lo que X en el dominio de enteros es el dominio a partir del cual el valor de x proporcionar valores para X en nuestro programa, por lo que podríamos decir que `int` es el tipo entero es

El tipo de la variable YX el mismo movimiento se traslada a los atributos como Bueno, los dominios típicos de los atributos son números enteros reales.

cadenas de estructuras booleanas, etc. ahora dados algunos dominios básicos como enteros caracteres reales y booleanos - podríamos construir nuevos dominios de los dominios dados utilizando operaciones matemáticas como el mapa de productos cruzados, etcétera, etcétera, por lo que incluso la intersección de la Unión, todos estos son permitido en estos dominios, hay dos ejemplos que proporcione aquí, uno es para una matriz, la otra es para una estructura, por lo que puede pensar en una sola dimensión matriz como un mapa de este dominio de números naturales al dominio de nuestro objetos de modo n a D , así que básicamente para cada número natural proporcionamos el objeto que se colocará en la ubicación de la matriz, por lo que si decimos que a es una matriz a $f1$

Los atributos en caso de que estos símbolos terminales lleven atributos que son necesarios para el cálculo de algunos atributos, entonces también serían los atributos de los símbolos terminales también serían parte del gráfico de dependencia.

El gráfico de dependencia ahora inicializa un qw o nuestro caso W con instancias de atributos B tal que B está en V y el grado de B es 0, lo que significa que no necesita cualquier otro atributo para calcular B , por lo que estos son que se puede calcular primero, por lo que estos son el nivel más bajo en el árbol de análisis vi antes, así que mientras W no será igual a Φ , esto es un algoritmo muy general por lo que es válido para cualquier atributo no circular, la gramática elimina algún atributo

```
int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}
main(){
    int p; int a[20], b[10];
    p = dot_prod(a,b);
}
```

Samples of static semantic checks in *main*

- Types of p and return type of *dot_prod* match
- Number and type of the parameters of *dot_prod* are the same in both its declaration and use
- p is declared before use, same for a and b

V2

Gramáticas libres de contexto cada símbolo del conjunto en Unión T se ha asociado con algunos atributos hay dos tipos de atributos heredados y sintetizados el mismo atributo no se puede heredar y sintetizar, pero podría ser que conozca varios atributos heredados en varios sintetizados asociado con cada símbolo, cada atributo toma un valor de un dominio como entero o real o producto cruzado de estos.

Asociamos reglas de cálculo de atributos con cada producción, de modo que reglas asociadas para el cálculo de atributos sintetizados de la mano izquierda lado no terminal y asociamos reglas para el cálculo de heredados atributos de los no terminales del lado derecho de la producción, por supuesto el aspecto más importante de una gramática de atributos es que es estrictamente



las reglas son estrictamente locales para cada producción P no hay efectos secundarios en las reglas, los atributos de los símbolos se evalúan sobre un árbol de análisis por haciendo pases sobre el árbol de análisis, es posible tener más de un pase y en cada pasada habría al menos un atributo calculado en los nodos por lo que estos atributos sintetizados se calculan de abajo hacia arriba de las hojas hacia arriba, así que básicamente son sintetizado a partir de los valores de atributo de estos hijos del nodo en el análisis

Las producciones se utilizan en el proceso de evaluación de atributos, la gramática del atributo para la evaluación de un número real a partir de su cadena de bits representación

Gramática genera cadenas binarias con el punto en el medio pero no tenemos indicación de cuál es su valor decimal el atributo que se supone que la gramática da el valor decimal de esta cadena de bits en particular generado por esta gramática, los no terminales C n R y B tienen solo un valor de atributo sintetizado que es del dominio de los reales mientras que el L no terminal tiene una longitud de atributo sintetizada y tiene un valor de atributo sintetizado por lo que la longitud es del dominio de los enteros y real listas del dominio de los números reales ¿por qué necesitamos dos atributos para L donde tiene uno es suficiente para cualquier LR y B para convencernos de que esto es requerido

El valor depende de L valor de punto y valor de punto entonces n longitud de punto L longitud de punto en este nodo depende de la longitud del punto L a continuación, mientras que el valor del punto l aquí

Gramática de atributo una diferente para el cálculo del número real de una representación de cadena de bits el modelo de estos dos ejemplos es que las oraciones en el idioma pueden ser las mismas gramáticas libres de contexto pueden volverse diferentes y, por lo tanto, el atributo Las gramáticas para el cálculo del mismo valor también pueden tener que volverse diferentes.

es generado por D yendo a TL de manera similar, XY flotante es generado por D yendo a TL pero estos dos juntos serán generados por d -list yendo a deliciosos punto y coma D y luego la lista d genera otra D , así es como se obtienen estas dos declaraciones

Atributo que es el nombre de la variable y el símbolo del terminal El identificador también tiene nombre como atributo sintetizado que no es nada.

una cadena de caracteres, entonces, ¿cómo funciona este atributo en particular? ya sabe a medida que avanzamos, por ejemplo, tomemos este ejemplo en KBC para que el tipo de las tres variables abc es int derecha a es de tipo int b es de tipo int y c es también de tipo int por lo que estos nombres ab y c son realmente generados en este nivel por el ID de producción va al identificador para asociar el nombre con su tipo, debemos de hecho, tome este int en particular y póngalo a disposición de la lista de nombres que se está generando, así que eso es precisamente lo que hacemos aquí la primera la producción primero en la lista d va a D o D menos va a lanzar el punto y coma D no tienen ningún cálculo asociado con ellos D va a TL tiene un atributo regla de cálculo, por lo que dice L tipo nulo que es el tipo de los nombres generado por L no es más que el tipo de punto T , por lo que el tipo de punto l se hereda y el punto T tipo se sintetiza, por lo que esta cosa son sintetizada y entrando en L ver un ejemplo con un árbol de análisis sintáctico en poco tiempo T yendo a int , así que aquí el tipo de punto T es entero, por lo que es muy simple de manera similar T va a flotar nos da un tipo de punto T igual al real L yendo a ID así que nuevamente L tiene el tipo como atributo heredado y debe pasarse al ID como un

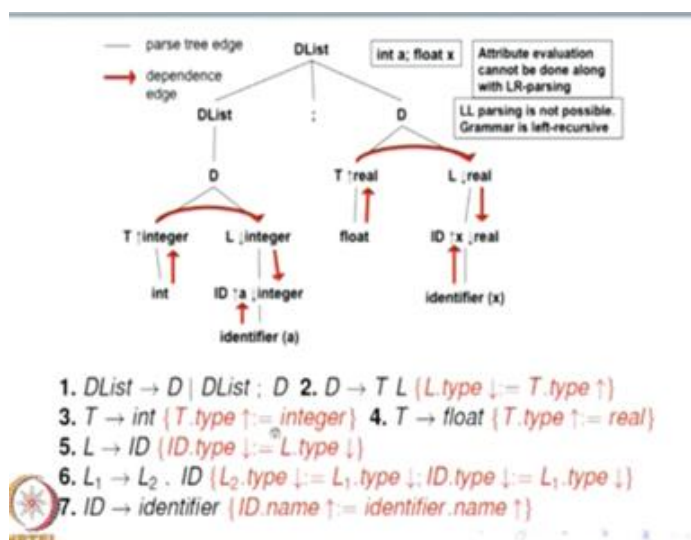


atributo heredado, por lo que el tipo de punto de ID es igual al tipo de punto L I1 que va a I2 ID de coma la L la información de tipo que se le da a I1 desde la parte superior se pasa a tanto I2 como ID aquí, por lo tanto, L 2 tipo de punto igual a L 1 tipo de punto e ID tipo de punto igual a L 1 tipo de punto ID que va al identificador por lo que ID el nombre del punto es el nombre del punto del identificador, por lo que en el nivel de ID hemos asociado ambos

el tipo y el nombre de esa variable en particular, por lo que una B y una C están etiquetadas con escriba un punto entero XYZ etiquetado con el tipo real en este ejemplo particular, por lo que Debe observar aquí que la notación para los atributos heredados es el atributo y luego la flecha hacia abajo y tenemos reglas de cálculo de reglas de atributos para los atributos heredados en el lado derecho de la producción por lo que L tiene en el atributo de encabezado, por lo que proporcionamos una regla de cálculo para él, pero no proporcionar cualquier regla de cálculo para el atributo sintetizado de T porque es en el lado derecho de esta producción en particular, mientras que T está en el lado izquierdo de esta producción, por lo que proporcionamos una regla de cálculo para ello así que esto y de manera similar L 1 que va a L 2 ID de coma L 2 tipo de punto se hereda, por lo que hay una regla de cálculo, cualquier tipo de punto está en el encabezado, por lo que hay una regla de computación

La evaluación de atributos se puede hacer muy fácilmente con tu pequeño o sabes

Analizadores sintácticos descendentes recursivos para ensayista se puede hacer con el análisis sintáctico lr e ll r Rd cualquiera de ellos, pero, por supuesto, si hay atributos heredados en general



Supongamos que el árbol de análisis sintáctico T con instancias de atributo no disminuido se da a nosotros y se supone que producimos un árbol de análisis T con un atributo consistente valores de los atributos de evaluación después del proceso de evaluación, así que básicamente hacemos un profundidad primero busque esa primera visita en el árbol de análisis para que DF visite con la raíz n es el punto de inicio para cada hijo M de n de izquierda a derecha evalúe los atributos heredados de M luego hacen un déficit en iam así que primero el entrada a un nodo evaluamos los atributos heredados y luego el déficit de fila que a su vez va hacia abajo en el árbol y después de completar la evaluación de todas las decisiones o dependientes de este M evaluamos esto en atributos de tamaño de N y luego volver al nivel superior, primero heredado y luego visitar el nodo y luego calcular el sintético que tributa al nodo y luego devolverlo Este es el procedimiento general



gramática es un conjunto de asociados o atributos que están asociados con él dos tipos de los atributos son posibles heredados y sintetizados y hay reglas asociadas con cada una de las producciones para calcular estos atributos para atributos sintetizados del lado izquierdo no terminales y heredados

Los atributos de los no terminales del lado derecho se proporcionan con reglas para calcular las reglas de atributos son, por supuesto, estrictamente locales a la producción y no tienen efectos secundarios por lo que ahora la clasificación de L atribuida en muertes gramáticas atribuidas si AG's son muy simples solo están sintetizados atributos y podemos utilizar cualquier estrategia de evaluación ascendente en una sola pasada para evaluar todos los atributos para que, debido a esta propiedad, se puedan combinar con el análisis sintáctico LR y, por lo tanto, reaccionar es muy útil con tales si G en el L

Gramáticas atribuidas, las dependencias van de izquierda a derecha de manera muy específica los atributos se pueden sintetizar y si se heredan tienen la siguiente limitación, por lo que en una producción $P \rightarrow X_1 X_2 \dots X_n$ supongamos que considere un atributo DX_i que se hereda y luego se excita dependen sólo de los elementos de aquí IFA que son los atributos heredados del lado izquierdo no terminal o los atributos de cualquiera de los símbolos a la izquierda de X es que k es igual a 1 a l menos 1 , por lo que nos concentramos en el SI G y elegía de una pasada en la que podemos hacer una evaluación de atributos con y l son ya analizando, por lo que la evaluación de atributos de las elegías es muy simple, solo hacemos una búsqueda profunda en el árbol de análisis que ya conoce, así que dado un árbol de análisis sintáctico con atributos no evaluados, la salida es un árbol de análisis sintáctico con valores de atributo consistentes,

DF llamada recursiva y después vista a todos los para que evalúe esto en los atributos de sociedad de n por lo que este es el algoritmo para la evaluación de tributos de gramáticas atribuidas reales

Las órdenes de evaluación pueden volverse un poco limitadas, restringidas, todas las Los pedidos no son posibles

Las restricciones se agregan al gráfico de dependencia de tributos como bordes implícitos y estas acciones se pueden agregar tanto a las gramáticas atribuidas yes como a gritar gramáticas atribuidas, por lo que en ese caso se convierten en ACTG y se eliminan

Así que en nuestro análisis semántico vamos a utilizar la TGS de una variedad de paso y de cursos SAT que siempre son de una pasada, así que aquí hay un SAT G para calculadoras de escritorio,

tabla productos y el parámetro es el nombre que es dólar uno, así que si el nombre está presente en la tabla de símbolos produce un puntero que apunta a que entrada en la tabla de símbolos y si el nombre no está en la misma luego introduce ese nombre en la tabla de símbolos y luego sabe que regresa el puntero a la nueva entrada en sp así que una vez que suceda podemos insertar el valor en la tabla de símbolos correspondiente a esa entrada de SP por lo que SP valor de puntero igual a dólar 3 introduce el valor de expresión y lo asocia con el punto SP que conoce con el nombre que es el que se señala a por SP y el valor devuelto por toda esta producción es el valor de la tercer símbolo que es expresión

Eso y cuando usamos un nombre buscamos ese nombre para que el símbolo del dólar uno busca ese nombre en la tabla de símbolos y devuelve un puntero si el nombre no está presente en la tabla



de símbolos se introduce en la tabla de símbolos y el valor predeterminado de 0 se inicializa en el campo de valor del nombre y un Laval o

el valor de la expresión en el lado izquierdo, por lo que estas son realmente acciones que no modifican ningún otro atributo de la gramática pero ciertamente son no cálculos de atributos, por eso es un ya sabes un sintetizado gramática de traducción de atributos SAT G, así que aquí vamos y continuamos con Otro ejemplo este ejemplo en realidad proporciona un cambio de gramática para el

LAG (notice the changed grammar)

1. $Decl \rightarrow DList\$$ 2. $DList \rightarrow D D'$ 3. $D' \rightarrow \epsilon \mid ; DList$
4. $D \rightarrow T L \{ L.type \downarrow := T.type \uparrow \}$
5. $T \rightarrow int \{ T.type \uparrow := integer \}$ 6. $T \rightarrow float \{ T.type \uparrow := real \}$
7. $L \rightarrow ID L' \{ ID.type \downarrow := L.type \downarrow; L'.type \downarrow := L.type \downarrow; \}$
8. $L' \rightarrow \epsilon \mid . L \{ L.type \downarrow := L'.type \downarrow; \}$
9. $ID \rightarrow identifier \{ ID.name \uparrow := identifier.name \uparrow \}$

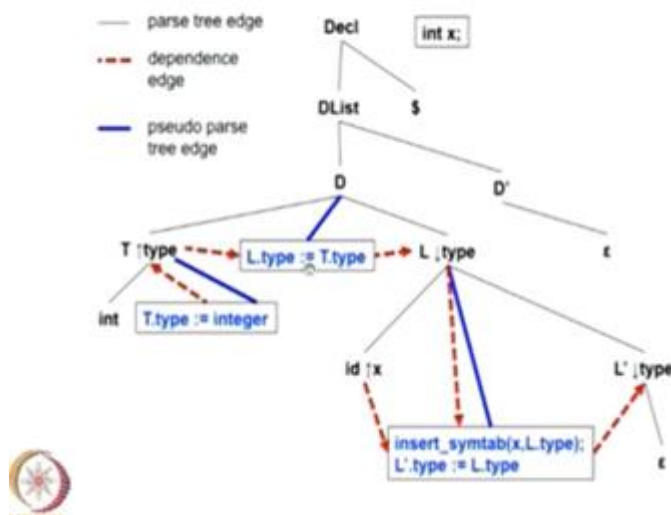
LATG (notice the changed grammar)

1. $Decl \rightarrow DList\$$ 2. $DList \rightarrow D D'$ 3. $D' \rightarrow \epsilon \mid ; DList$
4. $D \rightarrow T \{ L.type \downarrow := T.type \uparrow \} L$
5. $T \rightarrow int \{ T.type \uparrow := integer \}$ 6. $T \rightarrow float \{ T.type \uparrow := real \}$
7. $L \rightarrow id \{ insert_symtab(id.name \uparrow, L.type \downarrow);$
 $L'.type \downarrow := L.type \downarrow; \} L'$
8. $L' \rightarrow \epsilon \mid . \{ L.type \downarrow := L'.type \downarrow; \} L$



Declaraciones que vimos antes, ¿por qué deberíamos cambiar la gramática que cambiamos? porque la gramática anterior no era LL 1 y específicamente para una dama G, requieren una gramática que puede pasar por todos los analizadores son por un recursivo Descenso Parcells así que aquí el cambio está en eliminando la recursividad izquierda para que la declaración libere el acuerdo en dólares va a rehacer prime d prime va a epsilon semi coche punto y coma d-list resto de esto es lo mismo aquí también para la lista de identificadores quitamos la izquierda recursividad y la recursividad correcta para que L vaya a ID L Prime y L prime vaya a epsilon o coma L el resto de las producciones son las mismas y las Acciones semánticas en el caso de elegía siempre se pueden devolver al final de la producción simplemente porque el orden en el que escribimos las reglas semánticas dentro

La producción no es muy relevante, conoces las dependencias en realidad indicar el orden en el que deben realizarse los cálculos, por lo que dado el gramática de atributos de este tipo no será positivo esto es esto no tener acciones asociadas con él, así que no estaremos en posición de traducir esto en un programa directamente, mientras que si considera el L a TG



Usando un generador automático, la especificación latc diferente de la especificación L AG y AC g, la gramática aquí es la mismo que hemos tenido aquí hasta ahora la declaración de producción que va a eliminar de la lista la oferta en D va a DD Prime y D prime a épsilon o punto y coma d-list no hay acciones, así que consideremos este número para el papel de el cálculo se puede adjuntar al final, pero esto es la TG, por lo tanto inicialización de los atributos heredados de L cómo se hace solo antes de que pasemos la cadena generada por este año por lo que no escribirás igual al tipo de punto t, por lo que el tipo de punto T es un atributo sintetizado de T l tipo de punto es el atributo heredado del combustible, por lo que estamos calculando que justo antes de L, por lo que esta es la característica de una dama G calcula el atributos heredados de un no terminal justo antes de que el no terminal sea procesado, entonces tenemos t2 int que se puede adjuntar la regla, entonces realmente usted saber si no importa, entonces t2 float no hay otro orden posible T punto tipo igual a real ahora para la producción yel va a ID y L va a épsilon o punto y coma L tenemos extra o dos para la producción ul que va a l D tenemos insertar tabla de símbolos

por lo que la rutina se llama con el nombre de la identificador y el tipo de identificador por lo que el tipo de punto l es el atributo heredado del lado izquierdo no terminal ya es disponible, por lo que pasamos el nombre del punto de identificación y el tipo de punto l para que se inserte en el tabla de símbolos y el asociado que usted conoce, los campos apropiados están llenos y después de eso, los atributos de L prime se calculan justo antes de L prime, de modo que L prime atributo heredado del tipo de punto igual al tipo de punto L nuevamente desde el lado izquierdo de la producción, así que recuerde que podemos calcular los atributos heredados de la símbolo justo antes, pero los atributos sintetizados están disponibles desde abajo

las reglas de cálculo de atributos deben ejecutarse se muestra en rojo, así que si miras en esta posición tiene que ser este tipo de punto l igual al tipo de punto T ha sido insertado entre T y L para que la producción vaya a TL y el nodo se inserta aquí, veamos la gramática para comprobar si es correcta o incorrecto, de hecho es correcto, por lo que D va a T, luego la regla de cálculo y seguido de l, así que si simplemente hacemos la misma visita al DF en este árbol de análisis aumentado

Por ejemplo, incluso para esto, el atributo La regla de cálculo está entre ID y L prime L yendo al punto y coma de ID, sabes L prime, así que veamos que aquí, por ejemplo, ID y L prime, por lo que esta parte sabes que tenemos una acción semántica aquí, así que no hay punto y coma está aquí, por lo que ID



y L prime y la acción semántica se encuentra entre este la acción semántica se ha convertido en un pseudo nodo y los pseudo bordes se adjuntan a su padre L , por lo que si consideramos este árbol de análisis aumentado y simplemente realizamos una visita al DF y ejecute las reglas de cálculo de atributos en ese orden automáticamente los cálculos de atributos ocurren correctamente para que podamos ver que usted conozca la declaración, luego d -list y luego D luego T luego int para que este atributo obtenga

Calculado ahora cuando volvemos de int y una vez que lo hacemos subimos de nuevo a D y luego baje a este derecho, lo siento, así que después de eso, T y luego int y luego necesitamos visitar T tipo de punto igual a entero, por lo que calculamos el atributo de T y eso produce información de tipo aquí que sería int y ahora subimos a D y luego bajemos a esta acción para que no escribamos igual al tipo de punto T que prepara el atributo de para L para la próxima visita, luego subimos de nuevo y luego bajamos a L , por lo que este atributo ya está listo ahora vamos a ID y luego subimos de nuevo y luego bajamos a esta acción ahora podemos ver que el tipo de punto x y l están listos para esta acción semántica de insertar en la tabla de símbolos se puede llevar a cabo correctamente, entonces inicialmente l tipo de punto primo igual al tipo de punto L

Hacemos un déficit en el árbol de análisis, ejecute las acciones semánticas según sea necesario y eso atributos evaluados por lo que varios ATG para el mismo idioma de declaraciones es también se muestra aquí para comparar, por lo que d ir a TL es el mismo, por lo que la producción es la misma y solo sabemos que llamamos a una función antes de esto la producción mediante un procedimiento llamado tipo de parche, le diré por qué es necesario después de que bajemos un poco, por lo que t_2 en el papel de la computación es muy tipo de punto T simple igual a entero T para flotar es el tipo de punto T igual a real

Solo toma un atributo heredado, por lo que hay un parámetro que le corresponde a nuestro tipo de tipo y luego el cuerpo simplemente dice si mi token token es igual a ID para que sea la parte de análisis ahora la acción se introduce en el recursivo el analizador de descenso inserta la tabla de símbolos para que el nombre del punto de identificación no sea más que mi punto de token

El valor y el tipo de punto de grito no es más que el tipo de parámetro entrante, por lo que esto se ejecuta la función, luego obtenemos la siguiente y llamamos a L prime para que L prime dot tipo igual a l tipo de punto es la inicialización justo antes de L para que el código te hace saber ejecutado porque este tipo de información ya está disponible como el parámetro entrante que conoces no hay necesidad de otro cálculo aquí para que esté disponible automáticamente y se use aquí de lo contrario la próxima producción de Primera va a vaciar nuestra L prime yendo a la coma L así después de la coma hay un cálculo de atributo para que la función se amplíe L prime por lo que no hay retorno de ningún atributo sintetizado desde aquí atributo entrante parámetro entrante es el atributo heredado de nuestro tipo de tipo Entonces, si el token de my token dot es gamma, esta producción es aplicable luego obtenga el token y llame a L con el tipo de atributo heredado, por lo que el tipo es disponible como un parámetro entrante aquí, de lo contrario, si el token no es una coma entonces la producción aplicada es que L prime se vaciará, por lo que tenemos un valor nulo declaración punto y coma aquí la última declaración es d primo que va a vacío o en lugar de la última producción d prime que va a una lista d de punto y coma y no hay usted conocer los cálculos de tributos centrales aquí, por lo que es solo una parte parcial que es presente en la función del analizador sintáctico de descenso recursivo D prime, así que si mi token igual al punto y coma, luego obtenga la lista d de llamada token; de lo contrario, es la parte vacía, por lo que un punto y coma de declaración nula, así es como incrusta el analizador de descenso recursivo conoces la escena de acción semántica de un LED, él es la parte más importante para recuerde que aquí los atributos heredados se pasan como entrantes parámetros a una función y correspondientes a los no terminales y los atributos sintetizados son los resultados salientes de la



función correspondiente a ese no terminal, así que ahora veamos la versión SAT G de la gramática de evaluación de expresiones, por lo que es lo mismo que conoces la gramática con la producción especial e va a hacer que ID sea igual ae

un cálculo de tributo, pero cuando llegamos a la producción gritamos que vamos a dejar ID igual ae, por lo que hay varios puntos que deben tenerse en cuenta aquí: Primero, vamos a usar el nivel de anidamiento como el alcance de un nombre en particular.

por lo que el alcance se inicializa primero a cero para que cuando sepa que no hay otro posible anidamiento, el valor del alcance es cero y siempre que tengamos una nueva asociación deje que yo sea igual ae vamos a incrementar el alcance para que el alcance fuera inicializado a cero, este es el comentario ahora tan pronto como pasamos el ID de selección igual para saber que hay un nuevo alcance generado, así que incrementamos el alcance Inserte el nombre del punto de identificación en la tabla de símbolos con el nuevo alcance y el El valor es el que se asociará con el ID. El nombre del punto es e punto Val en ese nivel particular nuestro alcance, por lo que ahora el alcance es una variable global que está siendo manipulado en esta producción por lo que esta es una razón por la que se convierte en un SAT G en el La producción de EV dot Val es obviamente dot Val, así que eso no es un problema

En absoluto y ahora, después de esto, sabe que ahora tiene disponible la puntuación por símbolo tabla de modo que veremos el uso de e la tabla de símbolos en algún lugar aquí en la producción F va a ID pero en este momento supongamos que T tiene ha sido analizado y evaluado por lo que ha producido un valor utilizando la Asociación de Phi D a e, por lo que B dot val es un punto bueno ahora que hemos salido del alcance, así que elija ID igual a e rodilla, por lo que este es el final del alcance para el nombre de identificación asociado con la identificación

La declaración constante también se ingresa en la tabla de símbolos como un variable pero la bandera indica que es una constante pero de otro modo procesando Las declaraciones constantes no son muy diferentes de las de entregar variables.

Luego la otra cosa que debemos notar que cada identificador tiene varias piezas de información adjunta, por lo que este es el registro de información del tipo de identificado y toda esta información debe estar disponible en la tabla de símbolos también correspondiente a este nombre en particular, por lo que el nombre del identificador está disponible

Almacena el valor entero o real o tipo de error y este es el tipo de un identificador simple el tipo de un elemento de matriz



ejercicios prácticos del examen

Escribe una expresión regular para describir cada uno de las siguientes construcciones del lenguaje de programación:

a. Cualquier secuencia de tabulaciones y espacios en blanco (a veces llamados espacios en blanco)

/\t

\t

b. Comentarios en el lenguaje de programación c

// Comentar una línea

/ */ Comentar múltiples líneas*

. String constants (without escape characters) RESPUESTA: [A-Z][a-z]**

d. Floating-point numbers RESPUESTA: [-+]?[0-9]\.[0-9]+*



La siguiente gramática no es adecuada para un analizador sintáctico de predictivo de arriba hacia abajo. Identifique el problema y corrijalo reescribiendo la gramática. Demuestre que su nueva gramática satisface el LL (1) condition.

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

Se puede demostrar que la gramática cumple con la condición LL (1). Cómo demostrar que la nueva gramática es sin retroceso puede basarse en la definición de P8 Para cualquier no terminal que coincida con múltiples producciones $A, A \rightarrow \beta_1 \mid \beta_2 \mid \dots \beta_n$ PRIMERO + $(A \rightarrow \beta_i) \cap$ PRIMERO + $(A \rightarrow \beta_j) = \emptyset, \forall 1 \leq i, j \leq n, i \neq j$ Cualquier gramática con esta propiedad es sin retroceso El proceso de prueba es el siguiente Aquí PRIMERO + (1) se usa para denotar la producción PRIMERO + conjunto etiquetado 1

$$\text{PRIMERO}^+(9) = \{c\}$$

$$\text{PRIMERO}^+(8) = \{b\}$$

$$\text{PRIMERO}^+(8) \cap \text{PRIMERO}^+(9) = \emptyset$$

$$\text{PRIMERO}^+(5) = \{b\}$$

$$\text{PRIMERO}^+(6) = \{\epsilon, \text{eof}\}$$

$$\text{PRIMERO}^+(5) \cap \text{PRIMERO}^+(6) = \emptyset$$

$$\text{PRIMERO}^+(4) = \{c\}$$

$$\text{PRIMERO}^+(3) = \{a\}$$

$$\text{PRIMERO}^+(3) \cap \text{PRIMERO}^+(4) = \emptyset$$

$$\text{PRIMERO}^+(2) = \{b\}$$

$$\text{PRIMERO}^+(1) = \{a, c\}$$

$$\text{PRIMERO}^+(1) \cap \text{PRIMERO}^+(2) = \emptyset$$



SheepNoise

Una gramática libre de contexto, G , es un conjunto de reglas que describen cómo formar las expresiones *senSentence*. La colección de oraciones que se pueden derivar de G se llama una cadena de símbolos que se pueden derivar de la reglas de una gramática lenguaje definido por G , denotado G . El conjunto de lenguajes definidos por gramáticas libres de contexto se denomina conjunto de lenguajes libres de contexto. Un ejemplo puede ayudar. Considere la siguiente gramática, que llamamos SN:

SheepNoise \rightarrow *baa SheepNoise*

| *baa*

La primera regla o producción dice "*SheepNoise* puede derivar la palabra *baa* seguido de más *SheepNoise*". Aquí *SheepNoise* es una variable sintáctica una variable sintáctica utilizada en una gramática que representa el conjunto de cadenas que se pueden derivar de la gramática. Nosotros llame a dicha variable sintáctica un símbolo no terminal. Cada palabra en el idioma definido por la gramática es un símbolo terminal. La segunda regla dice una palabra que puede aparecer en una oración categoría. Las palabras se representan en una gramática por su categoría sintáctica "*SheepNoise* también puede derivar la cadena *baa*".

"Por lo tanto, *SheepNoise* \rightarrow + *baa*

y *SheepNoise* \rightarrow + *baa baa*"

Para derivar una oración en SN, comenzamos con la cadena que consta de un símbolo, *SheepNoise*. Podemos reescribir *SheepNoise* con la regla 1 o con la regla 2. Si reescribimos *SheepNoise* con la regla 2, la cadena se convierte en *baa* y no tiene más oportunidades de reescribir. La reescritura muestra que *baa* es una oración válida en L (SN). La otra opción, reescribir la cadena inicial con la regla 1, conduce a una cadena con dos símbolos: *baa SheepNoise*. Esta cadena tiene un no terminal restante; reescribirlo con la regla 2 conduce a la cadena *baa baa*, que es una oración en L (SN). Podemos representar estas derivaciones en forma de tabla:



Rule	Sentential Form
2	<i>SheepNoise</i> baa

Rewrite with Rule 2

Rule	Sentential Form
	<i>SheepNoise</i>
1	baa <i>SheepNoise</i>
2	baa baa

Rewrite with Rules 1 Then 2