review

The main work of lexical analysis

Classify program substrings according to token class

```
if (i == j)
    Z = 0;
else
    Z = 1;
```

\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;

Token class

Identifier
keyword
number
whitespace
special symbol
and so on

Token

<token class, lexeme>

(IF, “if”) (PLUS, “+”) (ID, “x”)

Example: foo=42

Summary

- An implementation must do two things:
  1. Recognize substrings corresponding to tokens
     - The *lexemes*
  2. Identify the token class of each lexeme

Questions: Regular Languages

What’s the function of Regular expressions?

Specify the lexical structure of programming languages

lexical structure=token classes

writing patterns to match a specific sequence of characters (also known as a string)

Questions:Regular Languages

- lexical structure=token classes

 Identifies: sequences of letters and digits beginning with a letter

 Numbers: numeric constants, such as 42, 3.14

 Reserve words: Fixed strings of characters that have special meaning in the language

 Special symbols: include arithmetic operations, assignment, equality and so on

Multiple Choices

Choose the regular languages that are equivalent to the given regular language: (0 | 1)*1(0 | 1)*

$\Sigma = \{0, 1\}$

- A (01 | 11)*(0 | 1)*
- B (0 | 1)*(10 | 11 | 1)(0 | 1)*
- C (1 | 0)*1(1 | 0)*
- D (0 | 1)*(0 | 1)(0 | 1)*

Lexical Specification

Extensions of regular expression to construct a full lexical specification on the programming language

Notations for regular expressions

- At least one: A^+ $\equiv$ AA^*
- Union: A | B $\equiv$ A + B
- Option: A? $\equiv$ A + ε
- Range: 'a' +' b' +...+' z' $\equiv$ [a-z]
- Excluded range:

 complement of [a-z] $\equiv$ [^a-z]

Partitioning a string into tokens

1. Write a rexp for the lexemes of each token class
   - Number = $digit^+$
   - Keyword = 'if' + 'else' + ...
   - Identifier = letter (letter + digit)*
   - OpenPar = '('
2. Construct R, matching all lexemes for all tokens

 R = Keyword + Identifier + Number + ...

 = $R_1 + R_2 + \ldots$

Partitioning a string into tokens

3. Let input be $x_1 \ldots x_n$

 For $1 \le i \le n$ check

 $x_1 \ldots x_i \in L(R)$

4. If success, then we know that

 $x_1 \ldots x_i \in L(R_j)$ for some j

5. Remove $x_1 \ldots x_i$ from input and go to (3)

Questions: Finite Automata

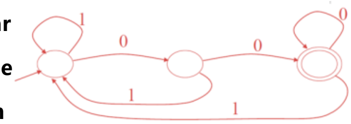
What's the function of finite automata?

Represent an abstracted version of a program
Implementation model for regular expression
Regular expression=specification
Finite automata=implementation
Example

Single Choice

In-Video Quiz **Finite Automata**

Select the regular language that denotes the same language as this finite automaton



A (0 + 1)*

B (1* + 0)(1 + 0)

C 1* + (01)* + (001)* + (000*1)*

D (0 + 1)*00

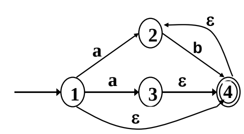
Questions Finite Automata

What's the difference between DFA and NFA?

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves

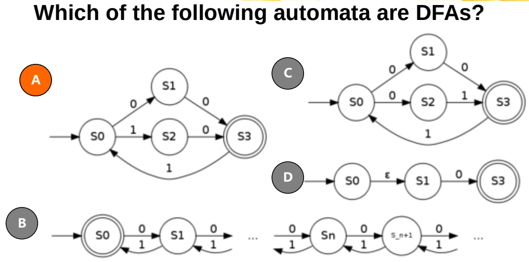
Questions Finite Automata

Give the paths for recognizing string abb



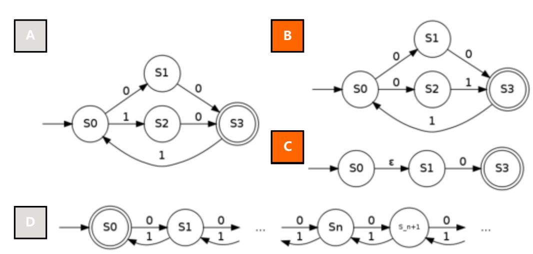
Single Choice

Quiz 1 Question 11
Which of the following automata are DFAs?

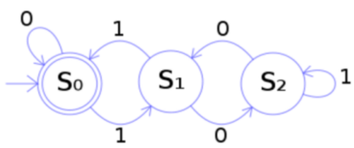


Multiple Choices

Quiz 1 Question 12
Which of the following automata are NFAs?



Making Useful Programs with DFAs



What strings will return "True" when passed through this DFA?

this automaton will return "True" for any binary number that represents a multiple of 3.

Chapter 3 Context-Free Grammars and Parsing

Study Goals:

Understand:

Context-free grammar, Derivation, Reduction, Parse tree, Abstract syntax tree

Know:

Hierarchy of grammar, Ambiguous Grammars

Main Content of Parsing Study

Specification of syntax structure :

Context-free grammar

Method of turning grammar rules into code for parsing

- Top-down parsing method
- Bottom-up parsing method

	Scanning	Parsing
Task	determining the structure of tokens	determining the syntax or structure of a program
Describing Tools	regular expression	context-free grammar
Algorithmic Method	represent by DFA	top-down parsing bottom-up parsing
Result Data Structure	liner structure	parser tree or syntax tree, they are recursive

3.1 The Parsing Process

Task of the parser

- Determine the syntactic structure of a program from the tokens produced by the scanner
- Either explicitly or implicitly, construct a parse tree or syntax tree that represent this structure.

Sequence of tokens → parser → Parse tree/ syntax tree

Interface of the parser

Input:

The parser calls a scanner procedure to fetch the next token from the input as it is needed during the parsing process

Output:

An explicit or implicit syntax tree needs to be constructed. Each node of the syntax tree includes the attributes needed for the remainder of the compilation process

Error Handling of Parser

Error Recover

Report meaningful error messages and resume the parsing as close to the actual error as possible(to find as many errors as possible)

Error repair

Infer a possible corrected code version from the incorrect version presented to it(This is usually done only in simple cases)

3.2 Context-Free Grammars

Function

A context-free grammar is a specification for the syntactic structure of a programming language

It is similar to regular expressions except that a context-free grammar involves recursive rules.

Example

context-free grammar for integer arithmetic expression

exp -> exp op exp | (exp) | number

op -> + | - | *

regular expression for number

number = digit digit*

digit = 0|1|2|3|4|5|6|7|8|9

3.2.1 Definition of Context-free Grammar

Definition

A context-free grammar $G= (V_T,V_N,P,S)$:

1. V_T is a set of terminals
2. V_N is a set of nonterminals, $V_N \cap V_T=\varphi$
3. P is a set of productions, or grammar rules, of the form $A\rightarrow \alpha$,where $A \in V_N$ and $\alpha \in (V_N \cup V_T)^*$
4. S is a start symbol , $S \in V_N$

Example

The grammar of simple arithmetic expressions

G=(V_T,V_N,P,S)

- V_T={num, +, -, *, /, (,) }
- V_N={exp, op}
- exp is the start symbol
- productions are:

exp -> exp op exp

exp -> (exp)

exp -> num

op -> +

op -> -

op -> *

op -> /

⇨

Explanation

1. V_T are the basic symbols from which strings are formed. Terminals are tokens
2. V_N are names for structures that denote sets of strings
3. The set of strings that start symbol "S" denotes is the language defined by the grammar
4. A production defines a structure whose name is to the left of the arrow. The layout of the structure is defined by the right of the arrow

⇦

5. "->" shows the left of the arrow cannot simply be replaced by its definition, as a result of the recursive nature of the definition
6. The form of productions(A→ α) is called Backus-Naur form(or BNF)

Notation Conventions

With the following conventions, we can only write productions of a grammar

Unless otherwise stated, the left side of the first production is the start symbol

Using lower-case letters to represent terminals

Using upper-case letters or name with <…> to represent nonterminals

If A->α_1, A-> α_2, …, A-> α_n are all productions with A on the left, we may write A-> α_1 | α_2 |… | α_n

Example

Grammar for simple arithmetic expressions can be written as:

E -> E O E | (E) | num

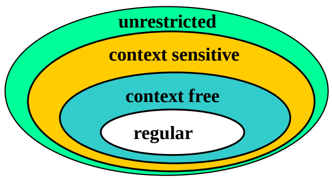
O -> + | - | * | /

Chomsky hierarchy of grammar

There are four kinds of grammars:

1. Unrestricted grammar (type 0)
2. Context sensitive grammar (type 1)
3. Context free grammar (type 2)
4. Regular grammar (type 3)

The Relationship of these four grammars:



Kind	Production	Explanation
unrestricted (type 0)	$\alpha \to \beta$ $V=V_N \cup V_T$ $\alpha \in V^+$, $\beta \in V^*$	there is no restriction to production
context sensitive (type 1)	$\alpha A \gamma \to \alpha \beta \gamma$, $\alpha,\gamma \in V^*$ $A \in V_N$, $\beta \in V^+$	A may be replaced only if α occurs before A and γ occurs after A
context free (type 2)	$A \to \beta$, $A \in V_N$, $\beta \in V^*$	A may be replaced anywhere,regardless of where A occurs
regular (type 3)	$A \to aB$ or $A \to a$, $A,B \in V_N$, $a \in V_T$	equivalent to regular expression

⇨

3.2.2 Derivation and Reduction

Function of derivation and reduction

Context-free grammar rules determine the set of syntactically legal strings of tokens

For example:Corresponding to grammar

exp->exp op exp | (exp) | number

op-> + | - | *

(34-3)*42 is a legal string

(34-3*42 is not a legal string

Grammar rules determine the legal strings of tokens by means of derivation or reduction

A derivation step and a reduction step

A-> βis a production of G , if there are strings v and w :v=αAγ,w=αβγ,where $\alpha,\gamma \in (V_N \cup V_T)^*$,we say there is a derivation step from v to w,or a reduction step form w to v,written as v=>w

- A derivation step is a replacement of a nonterminal by the right-hand side of the production
- A reduction step is a replacement of the right-hand side of production by the nonterminal on the left

Example

Grammar G: $S \to 0S1$, $S \to 01$

A derivation step

0S1	⇒00S11	($S \to 0S1$)
00S11	⇒000S111	($S \to 0S1$)
000S111	⇒00001111	($S \to 01$)
S	⇒0S1	($S \to 0S1$)

Derivation and Reduction

- The closure of => , α=>*β

  α=>*βif and only if there is a sequence of 0 or more derivation steps(n>=0),$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n$, such that $\alpha = \alpha_1$ and $\beta = \alpha_n$
- S=>*w, where $w \in V_T^*$ and S is the start symbol of G is called a derivation from S to w or a reduction from w to S

Example

exp->exp op exp | (exp) | num

op-> + | - | *

a derivation for (34-3)*42 is:

(1)exp	=>exp op exp	(exp->exp op exp)
(2)	=>exp op num	(exp->num)
(3)	=>exp * num	(op->*)
(4)	=>(exp)*num	(exp->(exp))
(5)	=>(exp op exp)* num	(exp->exp op exp)
(6)	=>(exp op num)*num	(exp->num)
(7)	=>(exp-num)*num	(op->-)
(8)	=>(num-num)*num	(exp->num)

3.2.3 The Language Defined by a Grammar

A sentential form of G

S is the start symbol of G,if S=>* α,α ∈ $(V_N \cup V_T)^*$,α is a sentential form of G

A sentence of G

w is a sentential form of G, if w contains only terminals, then w is a sentence of G

Example

G: S→0S1， S→01

S ⇒0<u>S</u>1 ⇒00<u>S</u>11 ⇒000<u>S</u>111 ⇒00001111

- S,0S1 ,00S11 ,000S111,00001111 are all sentential forms of G
- 00001111 is a sentence of G

The language defined by G,written as L(G)

L(G)={w $\in V_T^*$ |there exists a derivation S $\Rightarrow^*$w of G}

that is, L(G) is the set of sentences derivable from S

Example:

G:S→0S1,S→01

S=>0S1 => 00S11=> 0^3S1^3=> …=> $0^{n-1}S1^{n-1}$

=> 0^n1^n

L(G)={0^n1^n|n≥1}

The Relation between Grammar G and L(G)

- Given G, L(G) can be obtained by derivation

  Example

  G：E→E+T|T   T→T×F|F   F→(E)|a

  <u>E</u> ⇒<u>E</u>+T ⇒<u>T</u>+T ⇒<u>F</u>+T ⇒a+<u>T</u> ⇒a+<u>T</u>×F ⇒a+<u>F</u>×F ⇒a+a×<u>F</u> ⇒a+a×a

  L(G) are arithmetic expressions consisted of a,+, ×,(and)

- Given the description of L, we can design grammar for L

  Example

  A language consists of 0 and 1, every string of the language has the same number of 0 and 1

  Grammar for L is:

  A → 0B|1C
  B → 1|1A|0BB
  C → 0|0A|1CC

3.3 Parse Trees and Abstract Syntax Trees

3.3.1 Parse Trees

Function of parse trees

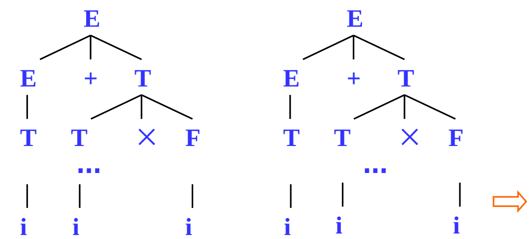
- A parse tree is a useful representation of the structure of a string of tokens
- Parse trees represent derivations visually

- Example: E→E+T|T   T→T×F|F   F→(E)|i

the derivation of “i+i×i” and the parse tree

E=>E+T=>E+T×F=>T+T×F =>… =>i+i×i

E=>E+T =>T+T =>T+T×F =>… =>i+i×i



Explanation:

- A parse tree of a string corresponds in general to many derivations of the string
- Parse trees abstract the essential features of derivations while factoring out superficial difference in order
- Derivations do not uniquely represent the structure of the strings they construct, while parse trees do

---

Definition of Parse tree

A parse tree over the grammar G is a rooted labeled tree with the following properties:

1. The root node is labeled with the start symbol S
2. Each leaf node is labeled with a terminal or with ε
3. Each nonleaf node is labeled with a nonterminal
4. If a node with label A ∈ V_N has n children with labels X1,X2,..,Xn(which may be terminals or nonterminals), then A->X1X2…Xn ∈P

Leftmost and Rightmost Derivation

Leftmost derivation

- A derivation in which the leftmost nonterminal is replaced at each step in the derivation
- It corresponds to a preorder traversal of the parse tree

Example

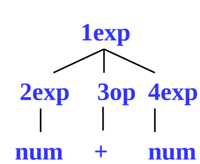
exp->exp op exp | (exp) | num

op-> + | - | *

- The leftmost derivation of “num+num” is:

exp=>exp op exp=>num op exp

=>num + exp=>num+num

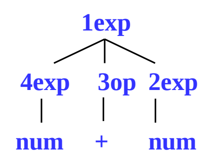


Rightmost derivation

- A derivation in which the rightmost nonterminal is replaced at each step in the derivation
- It corresponds to the reverse of a postorder traversal of the parse tree

❖ The rightmost derivation of "num+num" is:

exp=>exp op exp=>exp op num
=>exp + num=>num+num



- Leftmost and rightmost derivation are unique for the string they construct
- They are uniquely associated with the parse tree

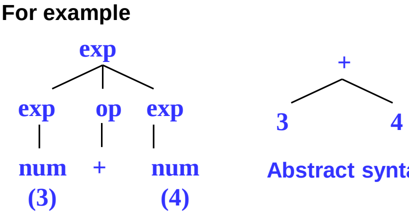
Relation of Parse tree and Derivation

- Each derivation gives rise to a parse tree
- Many derivations may give rise to the same parse tree
- Each parse tree has a unique leftmost and rightmost derivation that give rise to it
- Parse trees uniquely express the structure of syntax, as do leftmost and rightmost derivations, but not other derivations in general

3.3.2 Abstract Syntax Trees

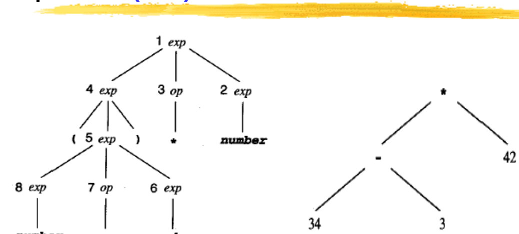
The need of abstract syntax tree

- A parse tree contains much more information than is absolutely necessary for a compiler to produce executable code
- For example



Abstract syntax tree

The parse tree and abstract syntax tree for expression (34-3)*42



Definition of Abstract Syntax trees

Abstract syntax trees represent abstractions of the actual string of tokens

Nevertheless they contain all the information needed for transition(represent precisely the semantic content of the string)

In more efficient form than parse trees

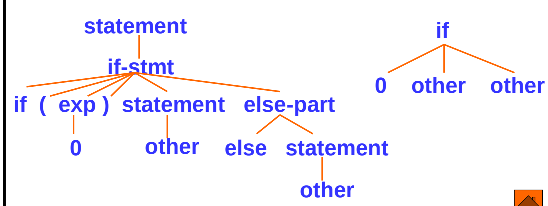
A parser will go through all the steps represented by a parse tree, but will usually only construct an abstract syntax tree

---

Example Grammar for statement

<statement> -> <if-stmt> | other
<if-stmt> -> if (<exp>) <statement> <else-part>
<else-part> -> else <statement> |ε
<exp> -> 0 | 1

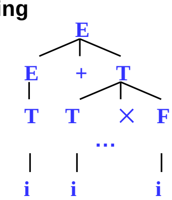
The parse tree and abstract syntax tree for string "if (0) other else other"



3.4 Ambiguity

In general, a string of tokens has one parse tree, which corresponds to more than one derivations of the string

Parse tree of string "i+i×i"



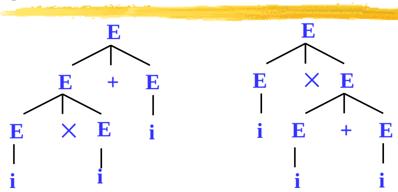
E=>E+T =>E+T×F =>T+T×F=>…=>i+i×i

E=>E+T =>T+T=>T+T×F =>…=>i+i×i

It is possible for some grammars to permit a string to have more than one parse tree(or leftmost/rightmost derivation)

For example:

Integer arithmetic grammar: E→E+E | E×E | (E) | i
String "i×i+i" has two different parse trees:



Corresponding to two leftmost derivations:

1: E ⇒ E+E ⇒ E×E+E ⇒ i×E+E ⇒ i×i+E ⇒ i×i+i

2: E ⇒ E×E ⇒ i×E ⇒ i×E+E ⇒ i×i+E ⇒ i×i+i

Ambiguity

A grammar G is ambiguous if there exists a string w ∈L(G) such that w has two distinct parse trees(or leftmost /rightmost derivations)

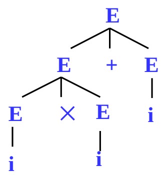
How to deal with ambiguity

Two basic methods are used to deal with ambiguities

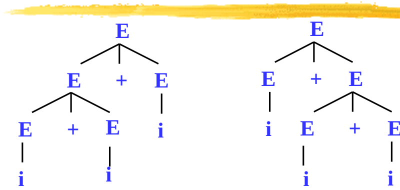
1. State a rule that specifies in each ambiguous case which of the parse trees is the correct one

Disambiguating rule:

- MUL has a higher precedence than ADD
- MUL and ADD are left associative



Integer arithmetic grammar: E→E+E|E×E|(E)|i
String "i+i+i" has two different parse trees:



Base on disambiguating rule, the first one is correct

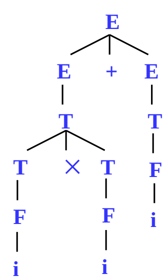
---

2. Change the grammar into a form that forces the construction of the correct parse tree

For example:E->E+E | E×E | (E) | i

- Add precedence
  Group the operators into groups of equal precedence, and for each precedence we must write a different rule:

E -> E+E | T   T -> T×T | F   F -> (E) | i
"i×i+i" is not ambiguous, the leftmost derivation for it is:
E=>E+E=>T+E=>T×T+E=>…=>i×i+i



but "i+i+i" is still ambiguous, which has two leftmost derivations:

E =>E+E=>E+E+E=>…

E =>E+E=>T+E=>F+E=>i+E

=>i+E+E=>…

- Add associativity: a left recursive rule makes its operators associate on the left, while a right recursive rule makes them associate on the right
  E -> E+T| T   T- > T×F | F   F -> (E)| i
  "i+i+i" is not ambiguous, the leftmost derivation for it is:
  E=>E+T=>E+T+T=>T+T+T=>…=>i+i+i