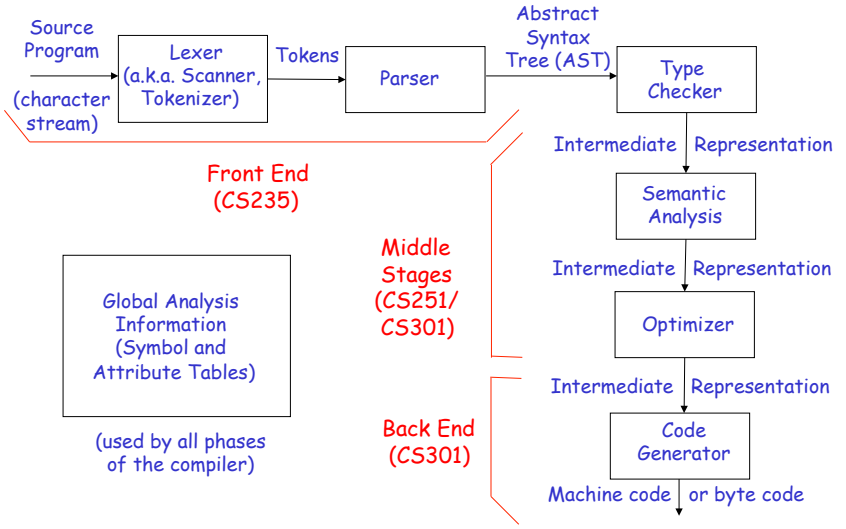
Introduction to Lexing and Parsing

Tuesday, November 29, 2011
Reading: Stoughton 4.3, Kozen 27

CS235 Languages and Automata

Department of Computer Science
Wellesley College

Compiler Structure



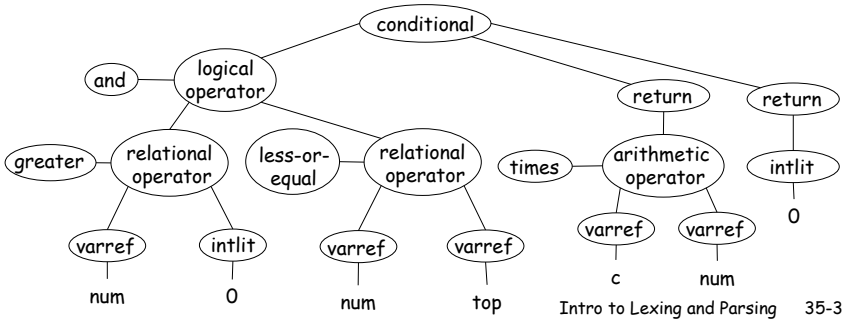
Front End Example

```
if (num > 0 && num <= top) { // Is num in range?
   return c*num
} else {return 0;}
```

Lexer (ignores whitespace, comments)

`if` `(` `num` `>` `0` `&&` `num` `<=` `top` `)` `{` `return` `c` `*` `num` `}`
`else` `{` `return` `0` `;` `}`

Parser (creates AST)



Lexical Analysis

Lexical analysis = breaking programs into tokens is the first stage of a compiler.

Program for lexical analysis has many names:
lexical analyzer, lexer, scanner, tokenizer

The structure of tokens can be specified by regular expressions.

The ML-Lex tool can automatically derive a lexical analyzer from a description of tokens specified by regular expressions.

There are many other tools for lexing, and parser generators (such as ANTLR and YACC) are equipped with lexing tools.

Sample English Description of Lexer Rules

An integer is a sequence of digits. A nonempty sequence of digits followed by E followed by a nonempty sequence of digits is scientific notation (e.g., 12E34 stands for 12×10^{34}).

An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different.

Certain names are reserved as **keywords** in the language and cannot be used as identifiers. E.g., Java keywords include **while**, **for**, **if**, **else**, **public**, **private**, **static**, **class**, **int**, **void**. ML keywords include **fun**, **let**, **in**, **end**, **if**, **then**, **else**.

If the input character stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token. Blanks, tabs, newlines, and comments (known collectively as **whitespace**) are ignored except as they serve to separate tokens. Some whitespace is required to separate otherwise adjacent identifiers, keywords, and constants.

Some ML-Lex Regular Expression Patterns

Pattern	Matches
"abc"	the literal string of characters abc
.	any character except newline
[a-zA-Z0-9]	any alphanumeric character
[^d-g]	any character except lowercase d,e,f,g
r_1r_2	r_1 followed by r_2, where r_1, r_2 are reg. exps.
$r_1\|r_2$	r_1 or r_2
r^*	zero or more rs, where r a reg. exp.
r^+	one or more rs
$r?$	zero or one rs
(r)	r (parens for grouping)
{*REName*}	regular expression with name *REName*

Regular Expressions for Some Tokens

"if"	if keyword
[a-zA-Z_][a-zA-Z0-9_]*	identifiers (variable names)
[0-9]+(E[0-9]+)?	integers

How should the following be split into tokens?

if	12
if89	1289
ifE89	12E89
ifEat34	12Eat34

Disambiguation rules:

Longest match. The longest initial substring of the input that can match any regular expression is taken as the next token.

Rule Priority. For a particular longest initial substring, the first regular expression that can match determines its token.

A SLiP Program

Here is a simple program in the straight-line programming language of Appel Ch. 1 (which I call SLiP):

```
sum := 5+3;
prod := (print (sum, sum-1), 10*sum);
print(prod);
```

Imagine that this is in the file test.slip.

We expect it to have the following tokens:

sum | := | 5 | + | 3 | ;

prod | := | (| print | (| sum | , | sum | - | 1 |) | ,

10 | * | sum |) | ;

print | (| prod |) | ; | EOF

How do we represent these tokens in SML?

A Token Data Type

```
datatype binop = Add | Mul | Sub | Div

datatype token = EOF
        |  ID of string
        |  INT of int
        |  OP of binop
        |  PRINT
        |  LPAREN |  RPAREN | COMMA | SEMI | GETS
```

token data type definition

```
sum := 5+3;
prod := (print (sum, sum-1), 10*sum);
print(prod);
```

Sample program

SML token list for sample program

```
[ID "sum", GETS, INT 5, OP Add, INT 3, SEMI,
 ID "prod", GETS, LPAREN, PRINT, LPAREN, ID "sum", COMMA, ID "sum",
 OP Sub, INT 1, RPAREN, COMMA, INT 10, OP Mul, ID "sum",  RPAREN, SEMI,
 PRINT, LPAREN, ID "prod", RPAREN, SEMI, EOF]
```


Some Token Operations

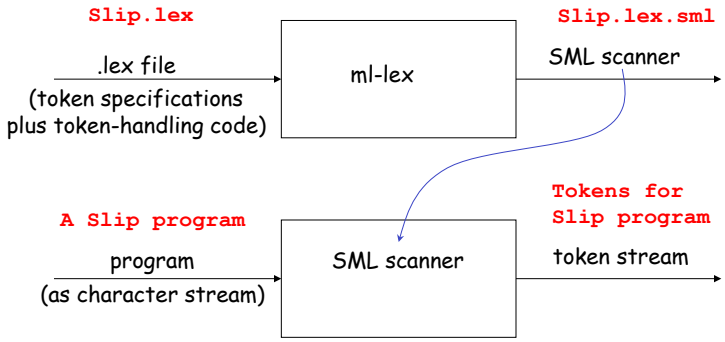
```
fun eof() = EOF

fun isEof(EOF) = true
  | isEof(_) = false

fun binopToString(Add) = "+"
  | binopToString(Sub) = "-"
  | binopToString(Mul) = "*"
  | binopToString(Div) = "/"

fun toString(EOF) = "[EOF]"
  | toString(ID(s)) = "[" ^ s ^ "]"
  | toString(INT(i)) = "[" ^ (Int.toString(i)) ^ "]"
  | toString(OP(opr)) = "[" ^ (binopToString(opr)) ^ "]"
  | toString(PRINT) = "[PRINT]"
  | toString(LPAREN) = "[(]"
  | toString(RPAREN) = "[)]"
  | toString(COMMA) = "[,]"
  | toString(SEMI) = "[;]"
  | toString(GETS) = "[:=]"
```


ml-lex: A Scanner Generator



Format of a .lex File

Header section with SML code
%%
Definitions of named regular expressions with form:
name=*regexp*
%%
Rules with pairs of token patterns & SML code having the form:
regexp => *SML-expression*

In *SML-expression*, the following special expressions may be used:

yytext	Stands for the string matching the expression
yypos	Character index of the first character of yytext in the input character stream
lex()	Ignores current token string and continues lexing
YYBEGIN <state>	Change state of lexer to <state>

Slip.lex Definitions and Rules

```
alpha=[a-zA-Z];
alphaNumUnd=[a-zA-Z0-9_];
digit=[0-9];
whitespace=[\ \t\n];
any= [^];
%%
"print" => (PRINT);
{alpha}{alphaNumUnd}*  => (ID(yytext));
{digit}+  => (INT(pluck(Int.fromString(yytext))));
"+" => (OP(Add));
"-" => (OP(Sub));
"*" => (OP(Mul));
"/" => (OP(Div));
"(" => (LPAREN);
")" => (RPAREN);
"," => (COMMA);
";" => (SEMI);
":=" => (GETS);
{whitespace} => (lex());
{any} => ((* Signal a failure exception when encounter unexpected character.
             A more flexible implementation might raise a more refined
              exception that could be handled. *)
          raise Fail("Slip scanner: unexpected character \"" ^ yytext ^ "\"")
```

Definitions

String matched by regular expression

Remove SOME from option type.

Discard current token and continue lexing

Rules

Using ml-lex to Generate a Scanner

```
[fturbak@sampras slip] ls -al Slip.lex.sml
ls: cannot access Slip.lex.sml: No such file or directory

[fturbak@sampras slip] ml-lex Slip.lex

Number of states = 27
Number of distinct rows = 10
Approx. memory size of trans. table = 1290 bytes

[fturbak@sampras slip] ls -al Slip.lex.sml
-rw-rw---- 1 fturbak fturbak 10277 2008-10-23 09:34 Slip.lex.sml
```

Contents of the file Slip.lex.sml

```
structure Mlex= struct
   structure UserDeclarations = struct ... end
   exception LexError
   structure Internal = struct ... end
   fun makeLexer yyinput = ...
   fun lex () = ...
end
```


Testing our Scanner

```
sum := 5+3;
prod := (print (sum, sum-1), 10*sum);
print(prod);
```

Sample program in file named "test.slip"

```
(* There are many details of how to interface the automatically
    generated lexer to SML that are not shown here.  Just assume
    that Scanner.fileToTokens lexes a file into a list of tokens. *)
- Scanner.fileToTokens "test.slip";
val it =
  [ID "sum", GETS,INT 5, OP Add, INT 3, SEMI, ID "prod", GETS,
    LPAREN, PRINT, LPAREN, ID "sum", COMMA, ID "sum",
    OP Sub, INT 1, RPAREN, COMMA, INT 10, OP Mul, ID "sum",
    RPAREN, SEMI, PRINT, LPAREN, ID "prod", RPAREN, SEMI] :
    Token.token list
```


Adding Line Comments

How to add line-terminated comments introduced by #?

```
sum := 5+3; # Set sum to 8
prod := (print (sum, sum-1), # First print sum and (sum-1),
        10*sum);                # then set prod to 10*sum
print(prod); # Finally print prod
```

The following ml-lex rule doesn't work. Why?

```
"#"{any}*"\n" => (lex() (* read a line comment *));
```

How can we fix it?

Adding Block Comments

How to add block (multi-line) comments delimited by { and } ?
(They needn't be nestable yet.)

```
sum := 5+3; # Set sum to 8
prod := (print (sum, sum-1), # First print sum and (sum-1),
        10*sum);             # then set prod to 10*sum
{ Comment out several lines:
  x := sum * 2;
  z := x * x; }
print(prod); # Finally print prod
```


Adding Nestable Block Comments

How to make block (multi-line) comments nestable ?

```
sum := 5+3; # Set sum to 8
prod := (print (sum, sum-1), # First print sum and (sum-1),
        10*sum);             # then set prod to 10*sum
{ Comment out several lines:
  x := sum * 2;
  { Illustrate nested block comments:
    y = prod + 3:}
  z := x * x; }
print(prod); # Finally print prod
```

Can't do this with regular expressions alone. (Why?)

Need some extra support! But any decent lexer should have this support, so there's no reason for PL designers to design languages without nested block quotes!!!

Using Lexer States for Nested Comments

```
(* Keeping track of nesting level of block comments *)
val commentNestingLevel = ref 0

fun incrementNesting() =
  (print "Incrementing comment nesting level";
   commentNestingLevel := (!commentNestingLevel) + 1)

fun decrementNesting() =
  (print "Decrementing comment nesting level";
   commentNestingLevel := (!commentNestingLevel) - 1)
%%
%s COMMENT;
alpha=[a-zA-Z];
alphaNumUnd=[a-zA-Z0-9_];
digit=[0-9];
whitespace=[\ \t\n];
any= [^];
%%
rules shown on next slide
```

Mutable state in SML modeled by mutable cells using **ref** and :=

New header functions

Declare a new state named COMMENT

Definitions

Lexer Rules for Nested Comments

```
<INITIAL>"print" => (PRINT);
<INITIAL>{alpha}{alphaNumUnd}*  => (ID(yytext));
<INITIAL>{digit}+  => (INT(pluck(Int.fromString(yytext))));
<INITIAL>"+" => (OP(Add));
<INITIAL>"-" => (OP(Sub));
<INITIAL>"*" => (OP(Mul));
<INITIAL>"/" => (OP(Div));
<INITIAL>"(" => (LPAREN);
<INITIAL>")" => (RPAREN);
<INITIAL>"," => (COMMA);
<INITIAL>";" => (SEMI);
<INITIAL>":=" => (GETS);
<INITIAL>"#".*"\n" => (lex() (* read a line comment *));
<INITIAL>"{" => (YYBEGIN COMMENT; incrementNesting(); lex());
<INITIAL>{whitespace} => (lex());
<COMMENT>"{" => (incrementNesting(); lex());
<COMMENT>"}" => (decrementNesting(); if (!commentNestingLevel) = 0 then
                 (YYBEGIN INITIAL; lex()) else lex());
<COMMENT>{any} => (lex());
{any} => ((* Signal a failure exception when encounter unexpected character.
          A more flexible implementation might raise a more refined
          exception that could be handled. *)
        raise Fail("Slip scanner: unexpected character \"" ^ yytext ^ "\""));
```

The Parsing Problem

Given a context-free grammar G and a string s:

- if $s \in$ Lang(G), return a parse tree in G that yields s;
- if $s \notin$ Lang(G), reject it.

Efficient solutions to this problem are important for analyzing both programming languages and natural languages.

The Parsing Problem: Examples

Example Grammar:

S → T | 0S1
T → % | 10T

Example Strings:

000111 011011
101010 001011
010101 011001

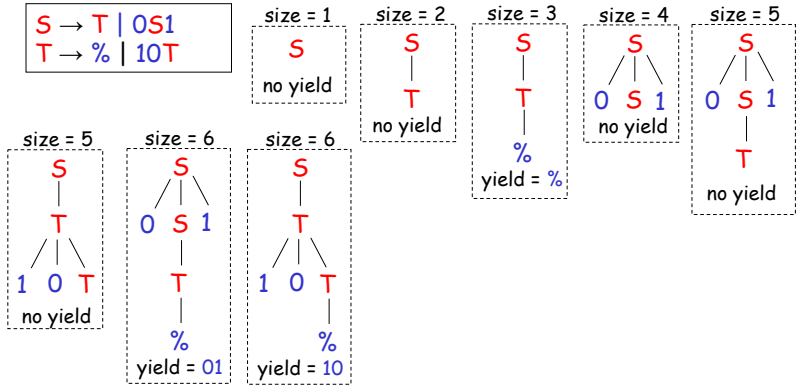
Goals For the Next Few Lectures

- Begin with some simple, but inefficient, parsing algorithms that work for all grammars.
- Study some more efficient approaches that work for various kinds of restricted grammars
- Build our way up to grammars suitable for automatic parser-generators.

Naive Algorithm : Top-Down Generate and Test

Idea: generate all parse trees top down from the start variable in order of size (# of nodes or height) until find one that yields s.

S → T | 0S1
T → % | 10T



How to know when to reject? Helps to have grammar in Chomsky Normal Form or Greibach Normal form. Why?

A Relevant Joke

An engineer, a physicist and a mathematician are staying in a hotel.

The engineer wakes up and smells smoke. He goes out into the hallway and sees a fire, so he fills a trash can from his room with water and douses the fire. He goes back to bed.

Later, the physicist wakes up and smells smoke. He opens his door and sees a fire in the hallway. He walks down the hall to a fire hose and after calculating the flame velocity, distance, water pressure, trajectory, etc. extinguishes the fire with the minimum amount of water and energy needed.

Later, the mathematician wakes up and smells smoke. He goes to the hall, sees the fire and then the fire hose. He thinks for a moment and then exclaims, "Ah, a solution exists!" and then goes back to bed.

(From http://www.math.utah.edu/~cherk/mathjokes.html)

Problems with Top-Down Generate and Test

Although it works for any grammar, top-down generate and test has big problems:

- Extremely inefficient approach to parsing.
- Repeat much of the same work for any string.
- Never take advantage of the structure of the string!

Working Bottom-Up: The Enzyme* Algorithm

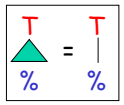
Idea: Build up all parse trees that can match substrings in the given string s. There are only a finite number of such substrings. Accept if there is a parse tree for s rooted at the start variable. (This is the algorithm presented in Stoughton 4.3.)

grammar *G*

S → T | 0S1
T → % | 10T

string *s* 001011

Step 1: Find all productions that yield % or some symbol in s.



Step 2: Based on productions in *G*, add parse trees that yield some substring of s.

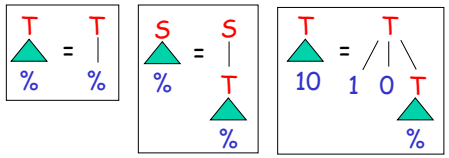
- "Enzyme" algorithm is my term. It is not standard. This is the algorithm used in Forlan and described in Stoughton 4.3. It is a variant of the classical Cocke-Kasami-Younger (CKY) dynamic programming algorithm for parsing described in Kozen Ch. 27.

Enzyme Example: The Next Round

grammar *G*

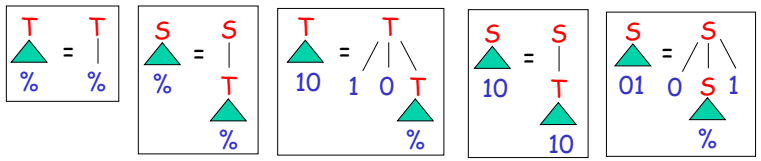
S → T | 0S1
T → % | 10T

string *s* 001011

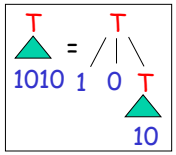


Enzyme Example Continued

grammar *G*: $S \to T \mid 0S1$, $T \to \% \mid 10T$ string *s*: 001011

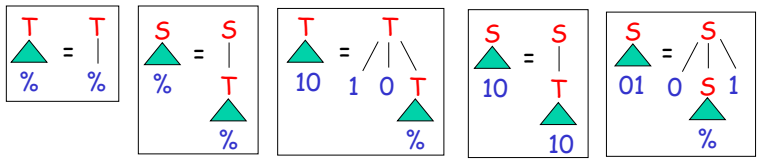


Note: we do not include this tree because 1010 isn't a substring of *s*.



Finishing The Enzyme Example

grammar *G*: $S \to T \mid 0S1$, $T \to \% \mid 10T$ string *s*: 001011



Flesh out the remaining trees in this example:

Problems with Enzyme Algorithm

- Although it works for any grammar *G* and takes input string *s* into account, the enzyme algorithm still can do much unnecessary work generating parse trees that do not appear in the final parse tree.
- The enzyme algorithm is related to the classic CKY algorithm. For that algorithm, which requires a CNF grammar, parsing a string w for a grammar G takes time $O(|G| \cdot |w|^3)$.
- It seems we can't do better than this for a general grammar. But we can develop more efficient algorithms for restricted grammars.

Backtracking Algorithm

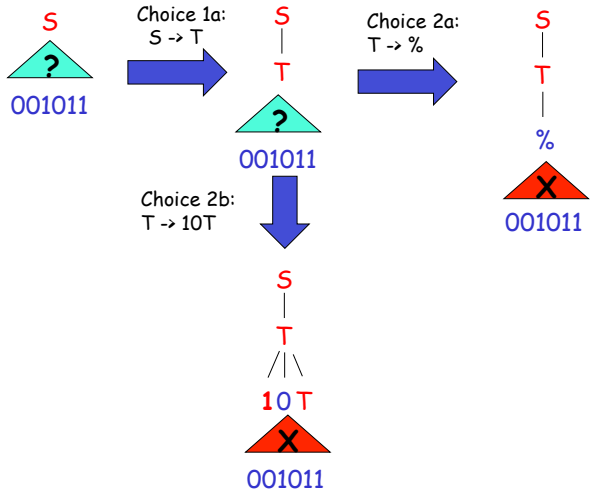
Idea:

- Start generating parse trees from start symbol.
- When there's a choice in production, make first choice and continue building trees whose prefix matches the target string.
- If there's ever a mismatch or unparsed symbols, go back to last choice point, and make the next choice.
- Continue until either (1) a parse tree for target string is found or (2) all possibilities have been explored and no tree is found.

Backtracking Example

grammar *G*

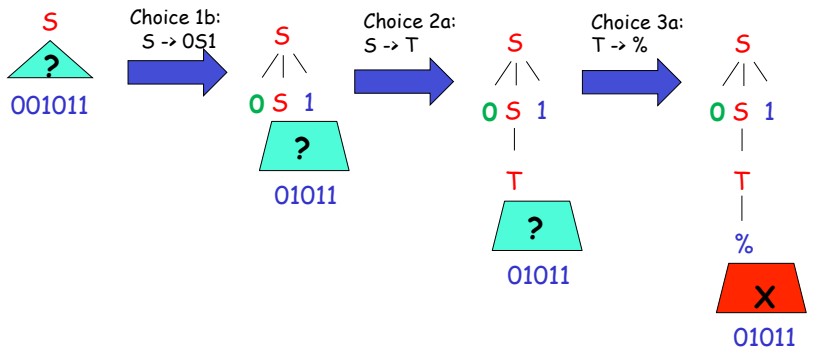
S → T | 0S1
T → % | 10T



Backtracking Example (Cont)

grammar *G*

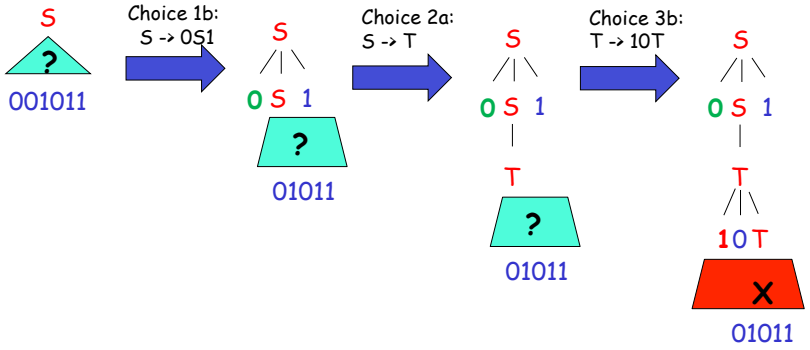
S → T | 0S1
T → % | 10T



Backtracking Example (Cont)

grammar *G*

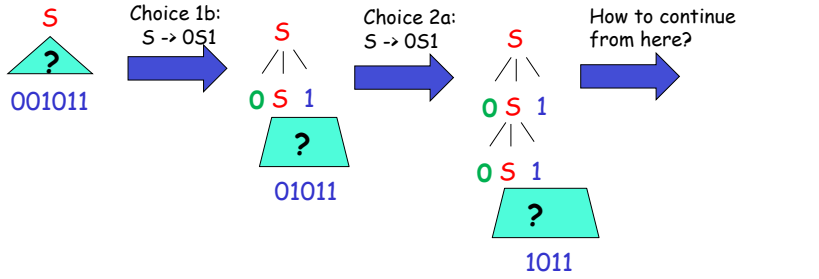
S → T | 0S1
T → % | 10T



Backtracking Example (Cont)

grammar *G*

S → T | 0S1
T → % | 10T



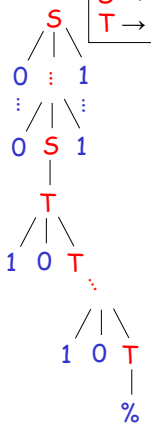
Towards A Deterministic Algorithm

Certain grammars (like our example) admit a deterministic parsing algorithm that depends on the structure of the grammar.

- Any initial sequence of 0s must be generated by S ⇒ 0S1 ⇒ 00S11 ⇒ 0… 0S1…1 ⇒ 0… 0T1…1
- If the first 1 is followed by a 0, it begins a sequence of 10s generated by T ⇒ 10T ⇒ 1010T ⇒ 10…10T ⇒ 10…10%
- If the first 1 is followed by a 1, it begins a sequence of 11s generated by the end of S ⇒ 0S1 ⇒ 00S11 ⇒ 0… 0S1…1 ⇒ 0… 0T1…1 in which the number of 1s must match the number of 0s in the initial sequence.

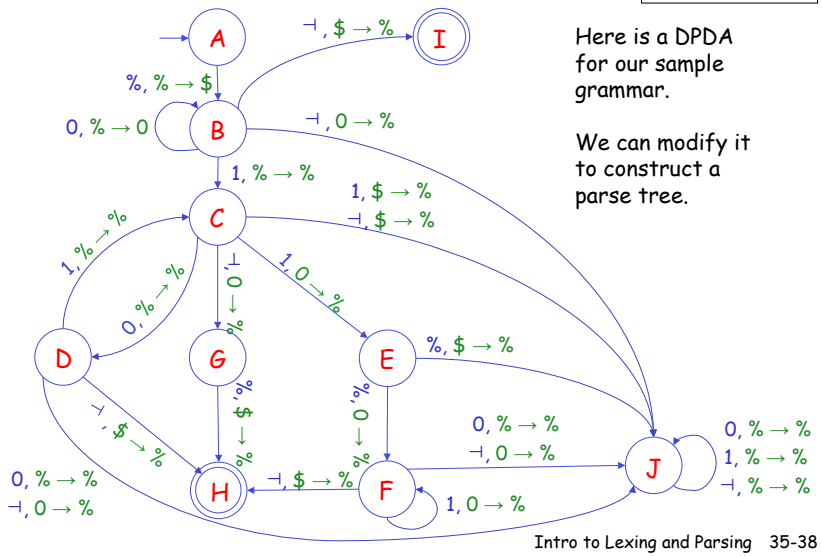
Key ideas: the **first** symbols that can be generated by a variable, what symbols can **follow** the subtree generated by a variable, **lookahead** past the current symbol.

S → T | 0S1
T → % | 10T



A Deterministic Parser

S → T | 0S1
T → % | 10T



Here is a DPDA for our sample grammar.

We can modify it to construct a parse tree.