Syntax and Parsing

COMS W4115

Prof. Stephen A. Edwards
Spring 2002
Columbia University
Department of Computer Science

Last Time

Administrivia

Class Project

Types of Programming Languages:

Imperative, Object-Oriented, Functional, Logic, Dataflow

This Time

Interpreters and Compilers

Structure of a Compiler

Lexical Analysis

Syntax

Parsing

The Compilation Process

Interpreters

$$\begin{array}{ccccc} & & \text{Source Program} & & \\ & & \downarrow & & \\ \text{Input} & \longrightarrow & \text{Interpreter} & \longrightarrow & \text{Output} \end{array}$$

Compilers

Source Program

$\downarrow$

Compiler

$\downarrow$

Input $\longrightarrow$ Executable Program $\longrightarrow$ Output

Structure of a Compiler

$\downarrow$ Program Text

Lexer

$\downarrow$ Token Stream

Parser

$\downarrow$ Abstract Syntax Tree

Static semantics (type checking)

$\downarrow$ Annotated AST

Translation to intermediate form

$\downarrow$ Three-address code

Code generation

$\downarrow$ Assembly Code

Compiling a Simple Program

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

What the Compiler Sees

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

```
i  n  t sp  g  c  d  (  i  n  t sp  a  , sp  i
n  t sp  b  ) nl  { nl sp sp  w  h  i  l  e sp
(  a sp  !  = sp  b  ) sp  { nl sp sp sp sp  i
f sp  (  a sp  > sp  b  ) sp  a sp  -  = sp  b
; nl sp sp sp sp  e  l  s  e sp  b sp  -  = sp
a  ; nl sp sp  } nl sp sp  r  e  t  u  r  n sp
a  ; nl  } nl
```

Text file is a sequence of characters

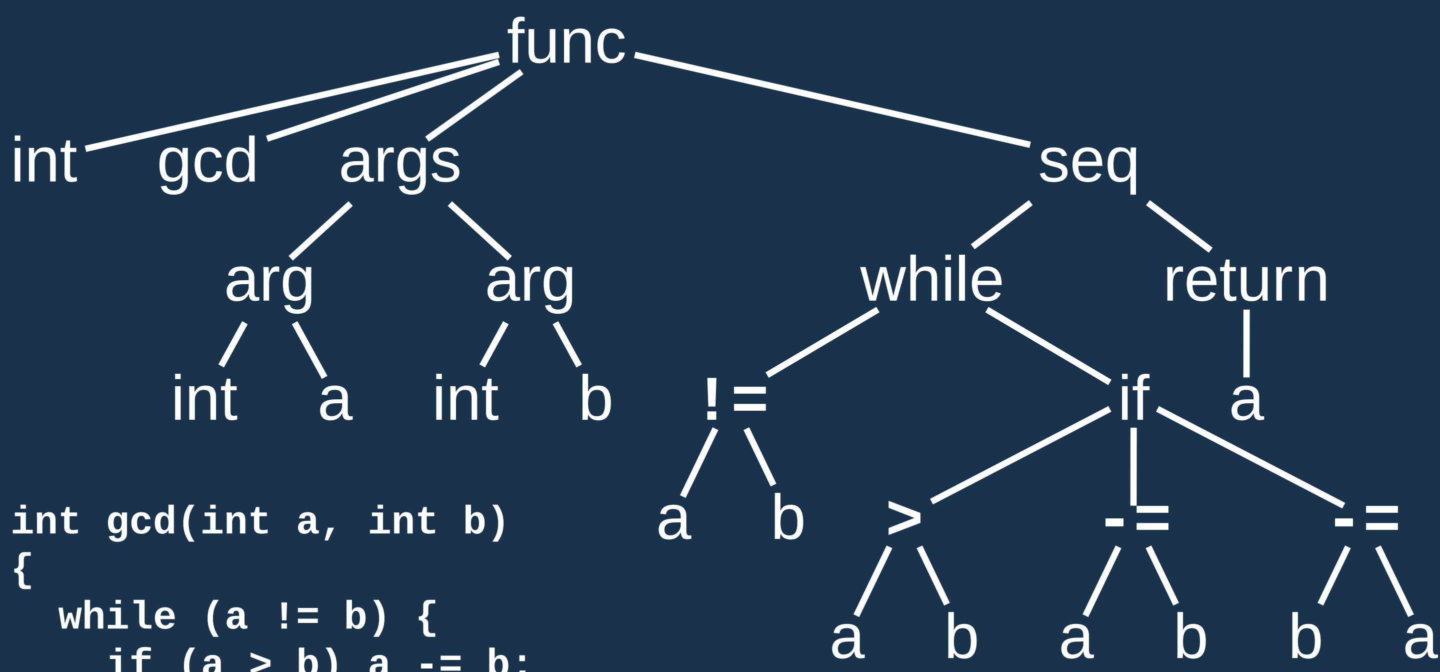
After Lexical Analysis

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

int	gcd	(	int	a	,	int	b	)	–	while	(	a		
!=	b	)	–	if	(	a	>	b	)	a	-=	b	;	
else	b	-=	a	;	˝	return	a	;	˝					

A stream of tokens. Whitespace, comments removed.

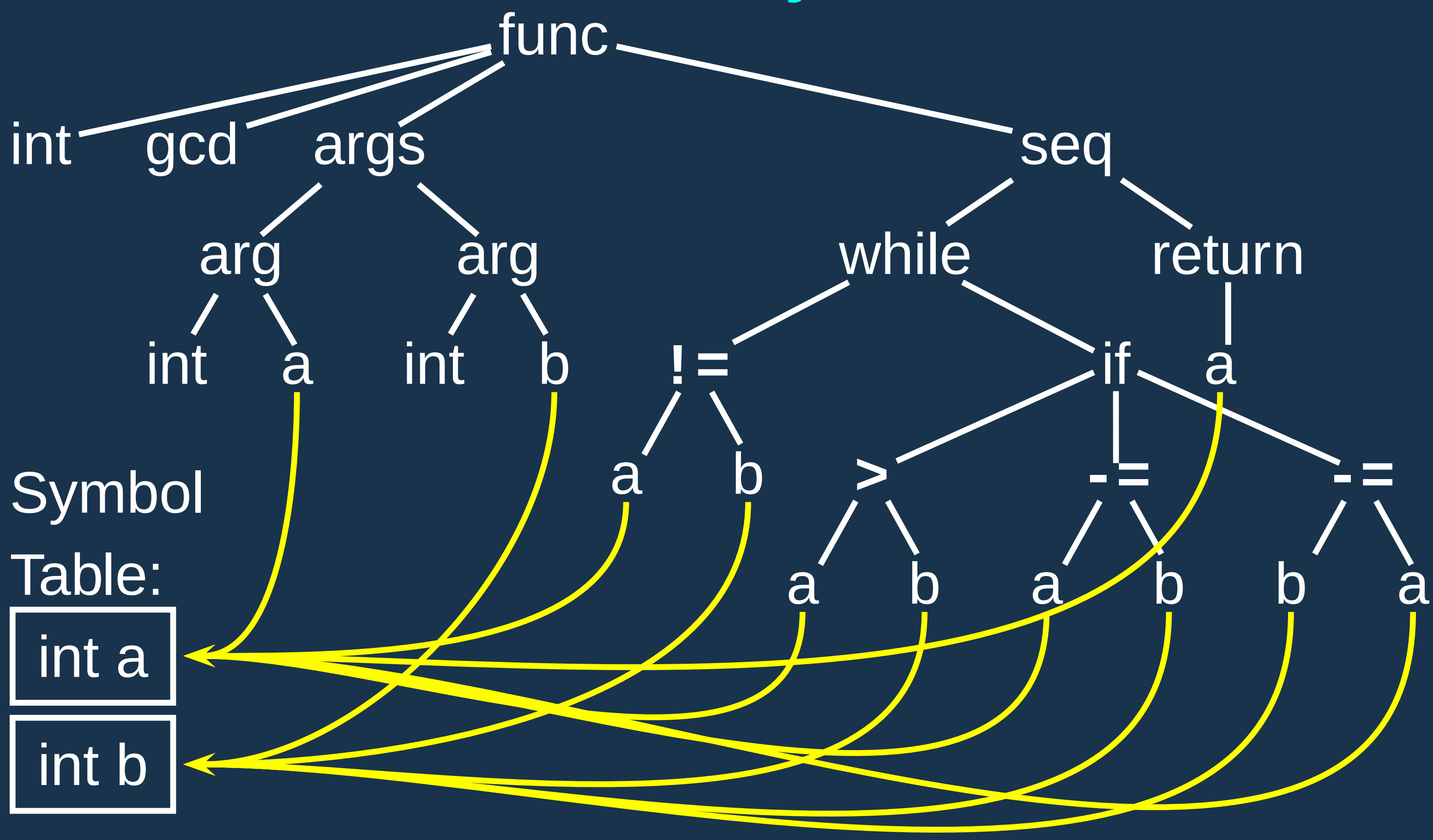
After Parsing




```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

Abstract syntax tree built from parsing rules.

After Semantic Analysis



Types checked; references to symbols resolved

After Translation into 3-Address Code

```
L0: sne   $1,  a, b
    seq   $0, $1, 0
    btrue $0, L1    % while (a != b)
    sl    $3,  b, a
    seq   $2, $3, 0
    btrue $2, L4    % if (a < b)
    sub   a,   a, b % a -= b
    jmp   L5
L4: sub   b,   b, a % b -= a
L5: jmp   L0
L1: ret    a
```

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

Idealized assembly language w/ infinite registers

After Translation to 80386 Assembly

```
gcd:   pushl %ebp              % Save frame pointer
       movl  %esp,%ebp
       movl  8(%ebp),%eax      % Load a from stack
       movl  12(%ebp),%edx     % Load b from stack
.L8:   cmpl  %edx,%eax
       je    .L3               % while (a != b)
       jle   .L5               % if (a < b)
       subl  %edx,%eax         % a -= b
       jmp   .L8
.L5:   subl  %eax,%edx         % b -= a
       jmp   .L8
.L3:   leave                   % Restore SP, BP
       ret
```

Lexical Analysis (Scanning)

Lexical Analysis (Scanning)

Goal is to translate a stream of characters

```
i  n  t sp  g  c  d  (  i  n  t sp
a  , sp  i  n  t sp  b
```

into a stream of tokens

ID	ID	LPAREN	ID	ID	COMMA	ID	ID
int	gcd	(	int	a	,	int	b

Each token consists of a token type and its text.

Whitespace and comments are discarded.

Lexical Analysis

Goal: simplify the job of the parser.

Scanners are usually much faster than parsers.

Discard as many irrelevant details as possible (e.g., whitespace, comments).

Parser does not care that the the identifer is “supercalifragilisticexpialidocious.”

Parser rules are only concerned with token types.

The ANTLR Compiler Generator

Language and compiler for writing compilers

Running ANTLR on an ANTLR file produces Java source files that can be compiled and run.

ANTLR can generate

- Scanners (lexical analyzers)
- Parsers
- Tree walkers

We will use all of these facilities in this class

An ANTLR File for a Simple Scanner

```
class CalcLexer extends Lexer;

LPAREN : ’(’ ;        // Rules for puctuation
RPAREN : ’)’ ;
STAR : ’*’ ;
PLUS : ’+’ ;
SEMI : ’;’ ;
protected             // Can only be used as a sub-rule
DIGIT : ’0’..’9’ ;    // Any character between 0 and 9
INT : (DIGIT)+ ;      // One or more digits

WS : (’ ’ | ’\t’ | ’\n’| ’\r’)     // Whitespace
     { $setType(Token.SKIP); } ;  // Action: ignore
```

ANTLR Specifications for Scanners

Rules are names starting with a capital letter.

A character in single quotes matches that character.

```
LPAREN : '(' ;
```

A string in double quotes matches the string

```
IF : "if" ;
```

A vertical bar indicates a choice:

```
OP : '+' | '-' | '*' | '/' ;
```

ANTLR Specifications

Question mark makes a clause optional.

```
PERSON : ("wo")? ’m’ (’a’|’e’) ’n’ ;
```

(Matches man, men, woman, and women.)

Double dots indicate a range of characters:

```
DIGIT : ’0’..’9’;
```

Asterisk and plus match “zero or more,” “one or more.”

```
ID : LETTER (LETTER | DIGIT)* ;
NUMBER : (DIGIT)+ ;
```

Kleene Closure

The asterisk operator (*) is called the Kleene Closure operator after the inventor of regular expressions, Stephen Cole Kleene, who pronounced his last name "CLAY-nee."

His son Ken writes "As far as I am aware this pronunciation is incorrect in all known languages. I believe that this novel pronunciation was invented by my father."

Scanner Behavior

All rules (tokens) are considered simultaneously. The longest one that matches wins:

1. Look at the next character in the file.
2. Can the next character be added to any of the tokens under construction?
3. If so, add the character to the token being constructed and go to step 1.
4. Otherwise, return the token.

How to keep track of multiple rules matching simultaneously? Build an automata.

Implementing Scanners Automatically

Regular Expressions (Rules)

↓

Nondeterministic Finite Automata

↓ Subset Construction

Deterministic Finite Automata

↓

Tables

Regular Expressions and NFAs

We are describing tokens with *regular expressions*:

- The symbol ϵ always matches
- A symbol from an alphabet, e.g., a, matches itself
- A sequence of two regular expressions e.g., $e_1 e_2$
  Matches e_1 followed by e_2
- An “OR” of two regular expressions e.g., $e_1 | e_2$
  Matches e_1 or e_2
- The Kleene closure of a regular expression, e.g., $(e)^*$
  Matches zero or more instances of e_1 in sequence.

Deterministic Finite Automata

A state machine with an initial state

Arcs indicate “consumed” input symbols.

States with double lines are accepting.

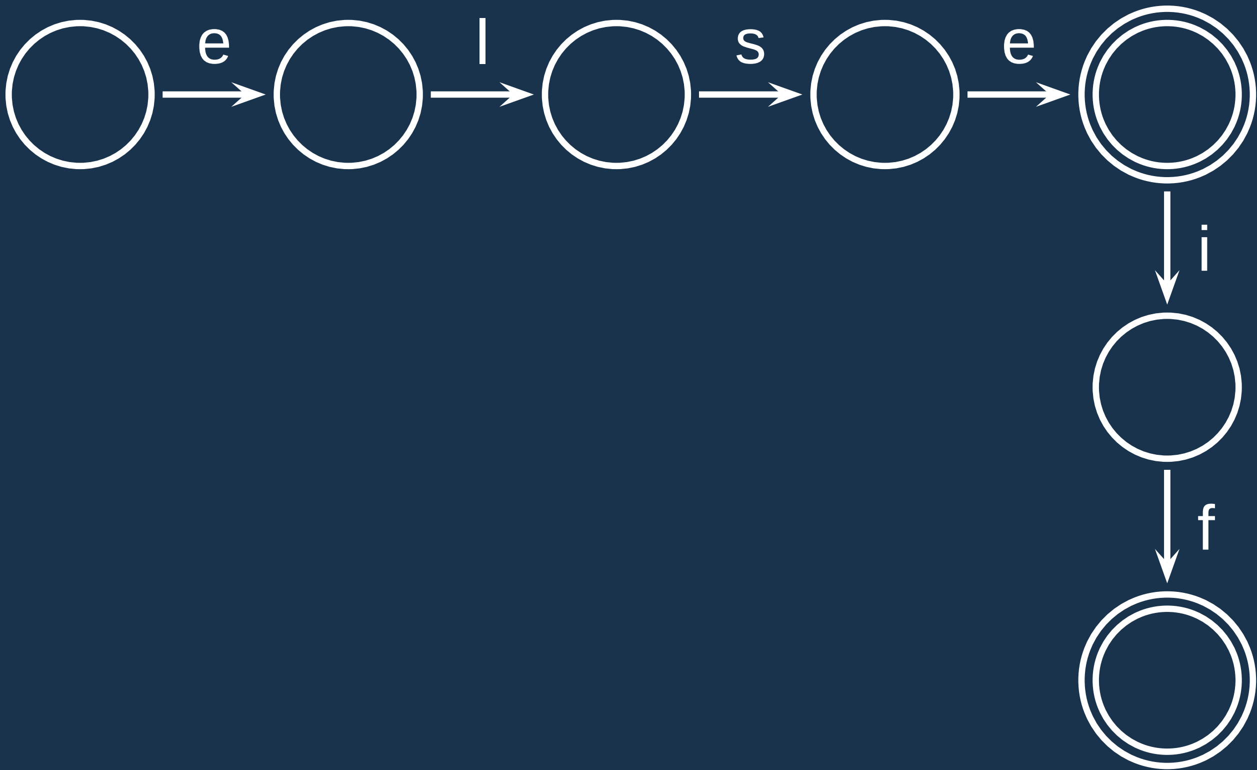
If the next token has an arc, follow the arc.

If the next token has no arc and the state is accepting, return the token.

If the next token has no arc and the state is not accepting, syntax error.

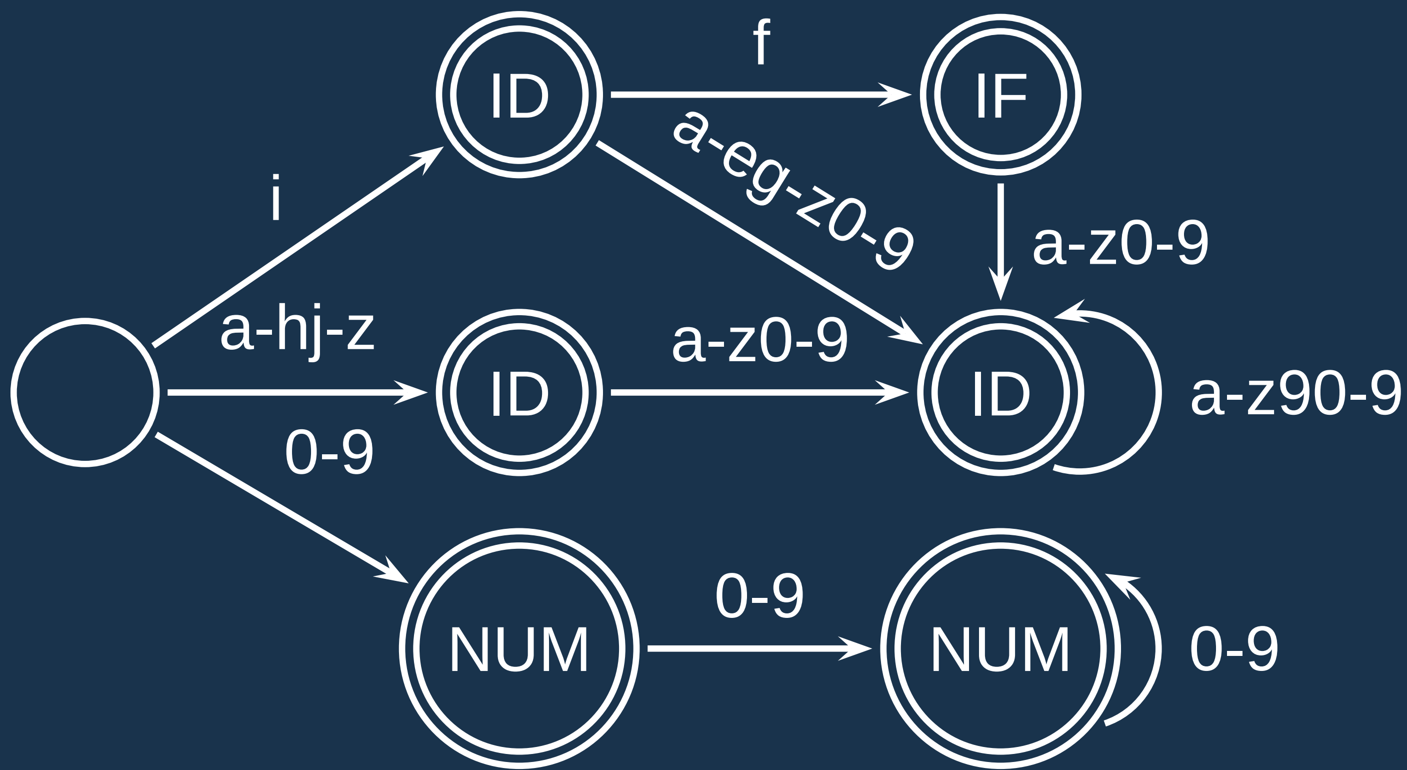
Deterministic Finite Automata

```
ELSE: "else" ;
ELSEIF: "elseif" ;
```



Deterministic Finite Automata

```
IF: "if" ;
ID: 'a'..'z' ('a'..'z' | '0'..'9')* ;
NUM: ('0'..'9')+ ;
```



Nondeterminstic Finite Automata

DFAs with ϵ arcs.

Conceptually, ϵ arcs denote state equivalence.

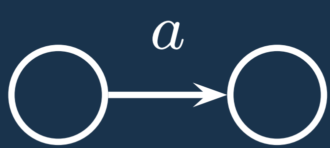
ϵ arcs add the ability to make nondeterministic (schizophrenic) choices.

When an NFA reaches a state with an ϵ arc, it moves into *every* destination.

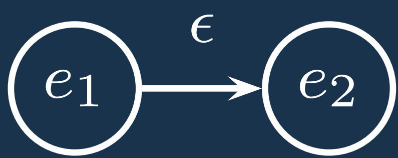
NFAs can be in multiple states at once.

Translating REs into NFAs

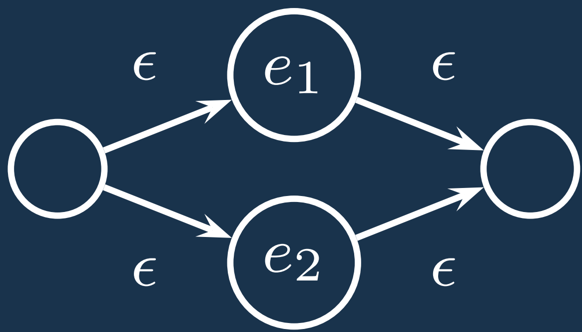
a



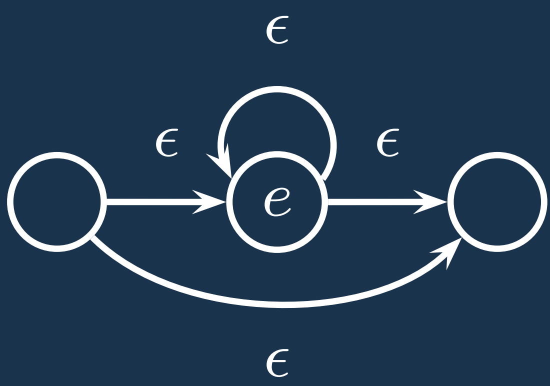
$e_1 e_2$



$e_1 | e_2$



$(e)^*$

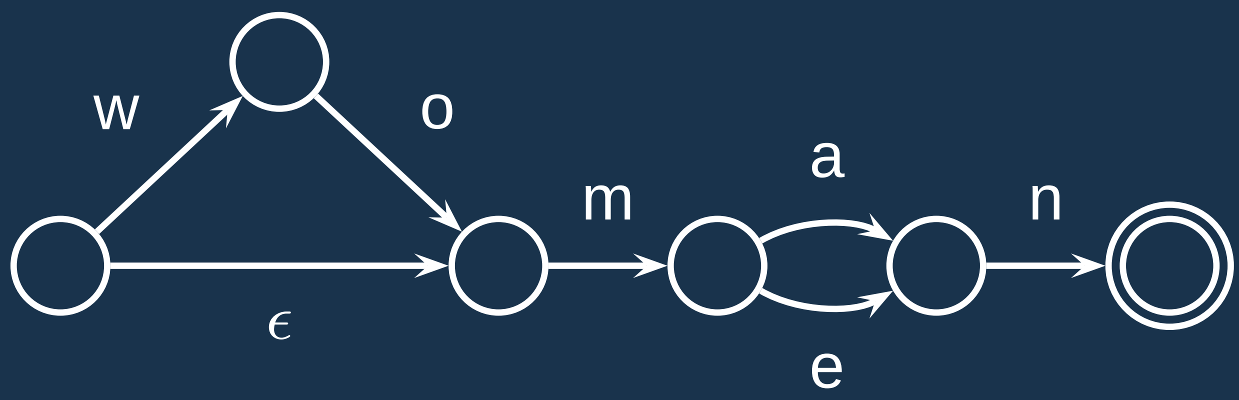


RE to NFAs

Building an NFA for the regular expression

$(wo|\epsilon)m(a|e)n$

produces

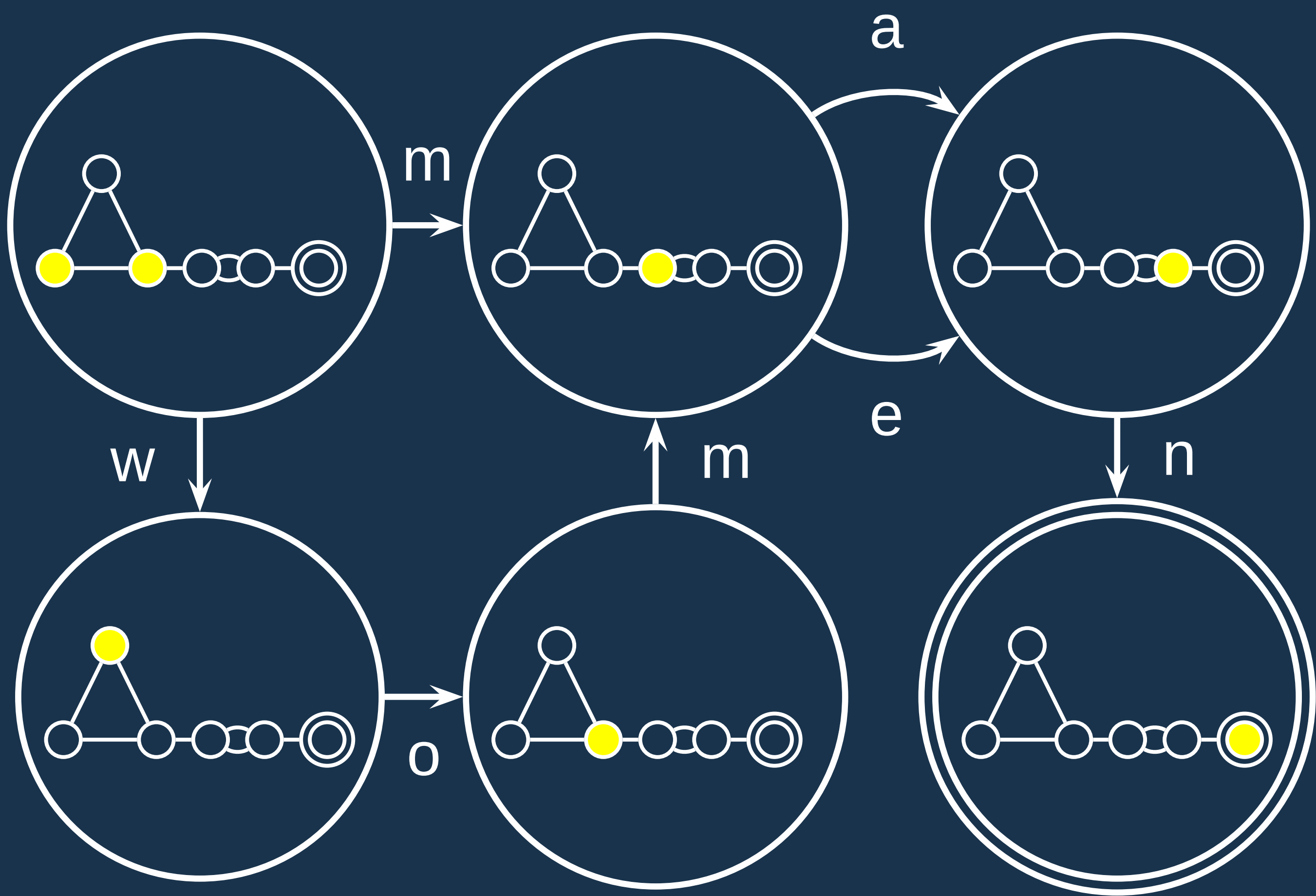


after simplification. Most ϵ arcs disappear.

Subset Construction

How to compute a DFA from an NFA.

Basic idea: each state of the DFA is a *marking* of the NFA



Subset Construction

An DFA can be exponentially larger than the corresponding NFA.

n states versus 2^n

Tools often try to strike a balance between the two representations.

ANTLR uses a different technique.

Free-Format Languages

Typical style arising from scanner/parser division

Program text is a series of tokens possibly separated by whitespace and comments, which are both ignored.

- keywords (**`if while`**)
- punctuation (**`, ( +`**)
- identifiers (**`foo bar`**)
- numbers (**`10 -3.14159e+32`**)
- strings (**`"A String"`**)

Free-Format Languages

Java C C++ Algol Pascal

Some deviate a little (e.g., C and C++ have a separate preprocessor)

But not all languages are free-format.

FORTRAN 77

FORTRAN 77 is not free-format. 72-character lines:

```
100   IF(IN .EQ. 'Y' .OR. IN .EQ. 'y' .OR.
     $   IN .EQ. 'T' .OR. IN .EQ. 't') THEN
```

$$\underbrace{\boxed{1}\cdots\boxed{5}}_{\text{Statement label}}\quad\underbrace{\boxed{6}}_{\text{Continuation}}\quad\underbrace{\boxed{7}\cdots\boxed{72}}_{\text{Normal}}$$

When column 6 is not a space, line is considered part of the previous.

Fixed-length line makes it easy to allocate a one-line buffer.

Makes sense on punch cards.

Python

The Python scripting language groups with indentation

```
i = 0
while i < 10:
    i = i + 1
    print i     # Prints 1, 2, ..., 10

i = 0
while i < 10:
    i = i + 1
print i         # Just prints 10
```

This is succinct, but can be error-prone.

How do you wrap a conditional around instructions?

Syntax and Langauge Design

Does syntax matter? Yes and no

More important is a language's *semantics*—its meaning.

The syntax is aesthetic, but can be a religious issue.

But aesthetics matter to people, and can be critical.

Verbosity does matter: smaller is usually better.

Too small can be a problem: APL is a compact, cryptic language with its own character set (!)

```
E←A TEST B;L
L←0.5
E←((A×A)+B×B)*L
```

Syntax and Language Design

Some syntax is error-prone. Classic FORTRAN example:

```
DO 5 I = 1,25  ! Loop header (for i = 1 to 25)
DO 5 I = 1.25  ! Assignment to variable DO5I
```

Trying too hard to reuse existing syntax in C++:

```
vector< vector<int> > foo;
vector<vector<int>> foo; // Syntax error
```

C distinguishes **>** and **>>** as different operators.

Keywords

Keywords look like identifiers in most languages.

Scanners do not know context, so keywords must take precedence over identifiers.

Too many keywords leaves fewer options for identifiers.

Langauges such as C++ or Java strive for fewer keywords to avoid “polluting” available identifiers.

Parsing

Parsing

Objective: build an abstract syntax tree (AST) for the token sequence from the scanner.

```
                         +
                        / \
2 * 3 + 4    ⇒         *   4
                      / \
                     2   3
```

Goal: discard irrelevant information to make it easier for the next stage.

Parentheses and most other forms of punctuation removed.

Grammars

Most programming languages described using a *context-free grammar*.

Compared to regular languages, context-free languages add one important thing: recursion.

Recursion allows you to count, e.g., to match pairs of nested parentheses.

Which languages do humans speak? I’d say it’s regular: I do not not not not not not not not not not understand this sentence.

Languages

Regular languages (t is a terminal):

$$A \longrightarrow t_1 \ldots t_n B$$

$$A \longrightarrow t_1 \ldots t_n$$

Context-free languages (P is terminal or a variable):

$$A \longrightarrow P_1 \ldots P_n$$

Context-sensitive languages:

$$\alpha_1 A \alpha_2 \longrightarrow \alpha_1 B \alpha_2$$

“$B \longrightarrow A$ only in the ‘context’ of $\alpha_1 \cdots \alpha_2$”

Issues

Ambiguous grammars

Precedence of operators

Left- versus right-recursive

Top-down vs. bottom-up parsers

Parse Tree vs. Abstract Syntax Tree

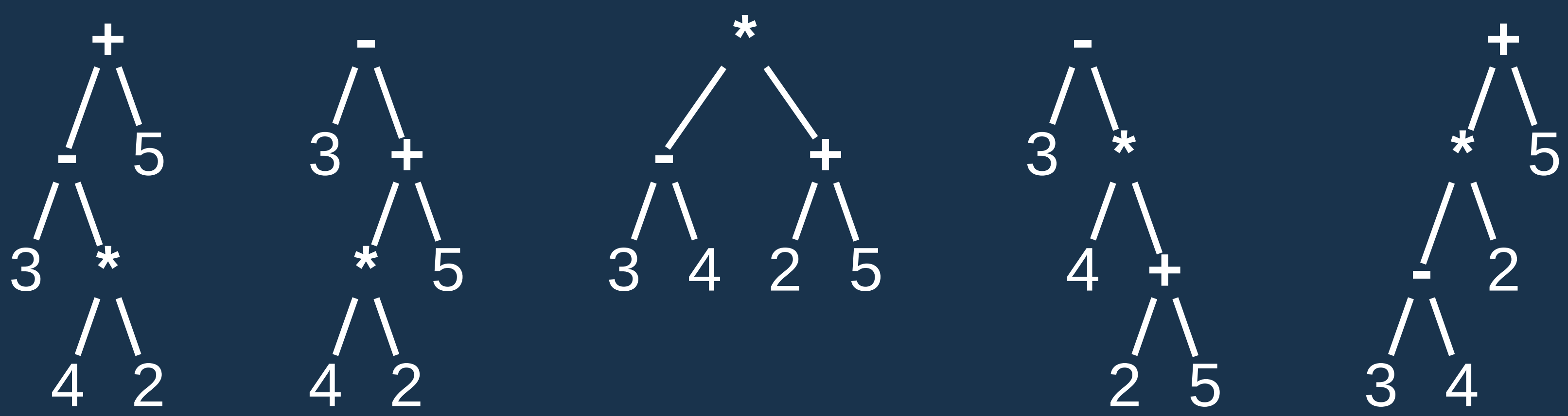
Ambiguous Grammars

A grammar can easily be ambiguous. Consider parsing

3 - 4 * 2 + 5

with the grammar

$$e \rightarrow e + e \,|\, e - e \,|\, e * e \,|\, e \,/\, e$$



Operator Precedence and Associativity

Usually resolve ambiguity in arithmetic expressions

Like you were taught in elementary school:

“My Dear Aunt Sally”

Mnemonic for multiplication and division before addition and subtraction.

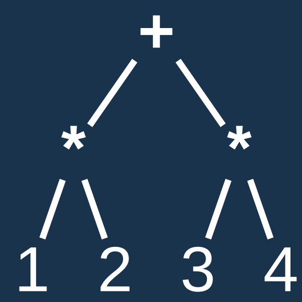
Operator Precedence

Defines how “sticky” an operator is.

`1 * 2 + 3 * 4`

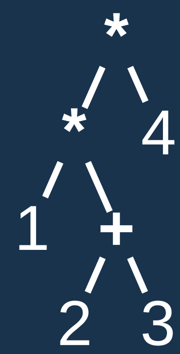
`*` at higher precedence than `+`:

`(1 * 2) + (3 * 4)`



`+` at higher precedence than `*`:

`1 * (2 + 3) * 4`



C’s 15 Precedence Levels

<table>
<tr><td>f(r,r,...)</td><td>a[i]</td><td>p->m</td><td>s.m</td></tr>
<tr><td>!b
++l
*p</td><td>~i
--l
&l</td><td>-i
l++
(type) r</td><td>
l--
sizeof(t)</td></tr>
<tr><td>n * o</td><td>n / o</td><td>i % j</td><td></td></tr>
<tr><td>n + o</td><td>n - o</td><td></td><td></td></tr>
<tr><td>i << j</td><td>i >> j</td><td></td><td></td></tr>
<tr><td>n < o</td><td>n > o</td><td>n <= o</td><td>n >= o</td></tr>
<tr><td>r == r</td><td>r != r</td><td></td><td></td></tr>
<tr><td>i & j</td><td></td><td></td><td></td></tr>
<tr><td>i ^ j</td><td></td><td></td><td></td></tr>
<tr><td>i | j</td><td></td><td></td><td></td></tr>
<tr><td>b && c</td><td></td><td></td><td></td></tr>
<tr><td>b || c</td><td></td><td></td><td></td></tr>
<tr><td>b ? r : r</td><td></td><td></td><td></td></tr>
<tr><td>l = r
l /= n
l |= i</td><td>l += n
l %= i
l <<= i</td><td>l -= n
l &= i
l >>= i</td><td>l *= n
l ^= i
</td></tr>
<tr><td>r1 , r2</td><td></td><td></td><td></td></tr>
</table>

Associativity

Whether to evaluate left-to-right or right-to-left

Most operators are left-associative

`1 - 2 - 3 - 4`

```
      -                     -
     / \                   / \
    -   4                 1   -
   / \                       / \
  -   3                     2   -
 / \                           / \
1   2                         3   4
```

((1 - 2) - 3) - 4   1 - (2 - (3 - 4))

left associative   right associative

Fixing Ambiguous Grammars

Original ANTLR grammar specification

```
expr
  : expr ’+’ expr
  | expr ’-’ expr
  | expr ’*’ expr
  | expr ’/’ expr
  | NUMBER
  ;
```

Ambiguous: no precedence or associativity.

Assigning Precedence Levels

Split into multiple rules, one per level

```
expr : expr ’+’ expr
     | expr ’-’ expr
     | term ;

term : term ’*’ term
     | term ’/’ term
     | atom ;

atom : NUMBER ;
```

Still ambiguous: associativity not defined

Assigning Associativity

Make one side or the other the next level of precedence

```
expr : expr ’+’ term
     | expr ’-’ term
     | term ;

term : term ’*’ atom
     | term ’/’ atom
     | atom ;

atom : NUMBER ;
```

Parsing Context-Free Grammars

There are $O(n^3)$ algorithms for parsing arbitrary CFGs, but most compilers demand $O(n)$ algorithms.

Fortunately, the LL and LR subclasses of CFGs have $O(n)$ parsing algorithms. People use these in practice.

Parsing LL(k) Grammars

LL: Left-to-right, Left-most derivation

k: number of tokens to look ahead

Parsed by top-down, predictive, recursive parsers

Basic idea: look at the next token to predict which production to use

ANTLR builds recursive LL(k) parsers

Almost a direct translation from the grammar.

A Top-Down Parser

```
stmt : 'if' expr 'then' expr
     | 'while' expr 'do' expr
     | expr ':=' expr ;

expr : NUMBER | '(' expr ')' ;
```

AST stmt() –

switch (next-token) –

case ”if” : match(”if”); expr(); match(”then”); expr();

case ”while” : match(”while”); expr(); match(”do”); expr();

case NUMBER or ”(” : expr(); match(”:=”); expr();

″ ″

Writing LL(k) Grammars

Cannot have left-recursion

```
expr : expr ’+’ term | term ;
```

becomes

AST expr() –

switch (next-token) –

case NUMBER : expr(); /* Infinite Recursion */

Writing LL(1) Grammars

Cannot have common prefixes

```
expr : ID '(' expr ')'
     | ID '=' expr
```

becomes

AST expr() –

switch (next-token) –

case ID : match(ID); match('('); expr(); match(')');

case ID : match(ID); match('='); expr();

Eliminating Common Prefixes

Consolidate common prefixes:

```
expr
  : expr '+' term
  | expr '-' term
  | term
  ;
```

becomes

```
expr
  : expr ('+' term | '-' term )
  | term
  ;
```

Eliminating Left Recursion

Understand the recursion and add tail rules

```
expr
  : expr ('+' term | '-' term )
  | term
  ;
```

becomes

```
expr : term exprt ;
exprt : '+' term exprt
      | '-' term exprt
      | /* nothing */
      ;
```

Using ANTLR’s EBNF

ANTLR makes this easier since it supports **`*`** and **`-`**:

```
expr : expr ’+’ term
     | expr ’-’ term
     | term ;
```

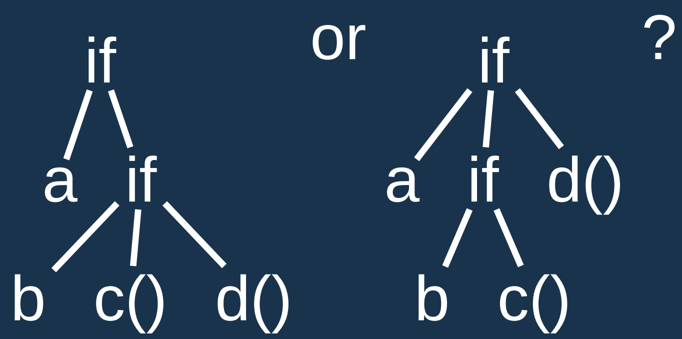
becomes

```
expr : term (’+’ term | ’-’ term)* ;
```

The Dangling Else Problem

Who owns the *else*?

```
if (a) if (b) c(); else d();
```



Grammars are usually ambiguous; manuals give disambiguating rules such as C’s:

> As usual the “else” is resolved by connecting an else with the last encountered elseless if.

The Dangling Else Problem

```
stmt : "if" expr "then" stmt iftail
     | other-statements ;

iftail
   : "else" stmt
   | /* nothing */
   ;
```

Problem comes when matching “iftail.”

Normally, an empty choice is taken if the next token is in the “follow set” of the rule. But since “else” can follow an iftail, the decision is ambiguous.

The Dangling Else Problem

ANTLR can resolve this problem by making certain rules “greedy.” If a conditional is marked as greedy, it will take that option even if the “nothing” option would also match:

```
stmt
  : "if" expr "then" stmt
     ( options {greedy = true;}
     : "else" stmt
     )?
  | other-statements
  ;
```

The Dangling Else Problem

Some languages resolve this problem by insisting on nesting everything.

E.g., Algol 68:

```
if a < b then a else b fi;
```

“fi” is “if” spelled backwards. The language also uses do–od and case–esac.

Bottom-up Parsers

Regular languages can be matched using finite automata.

Context-free languages can be matched with pushdown automata (have a stack).

Operation of a bottom-up parser:

- Maintain a stack of tokens and rules
- Push each new token onto this stack (“shift”)
- When the top few things on the stack match a rule, replace them (“reduce”)

Used by yacc, bison, and other parser generators.

Parses more languages, but error recovery harder.

Bottom-up Parsing

```
E : T '+' E | T ;
T : int '*' T | int ;
```

stack	input	action
	int * int + int	shift
int	* int + int	shift
int *	int + int	shift
int * int	+ int	reduce T : int
int * T	+ int	reduce T : int '*' T
T	+ int	shift
T +	int	shift
T + int		reduce T : int
T + T		reduce E : T
T + E		reduce E : T '+' E
E		reduce E : T '+' E

Parsing Techniques

Much theory has been developed about languages and parsing algorithms.

Could easily fill a term.

Fortunately, you don’t need to know all the technical details to build an effective parser using tools.

Just know about tools such as ANTLR, lex, flex, yacc, Bison, JLex, CUP, etc.

Statement separators or terminators?

C uses ; as a statement terminator.

```
if (a<b) printf("a less");
else {
  printf("b"); printf(" less");
}
```

Pascal uses ; as a statement separator.

```
if a < b then writeln('a less')
else begin
  write('a'); writeln(' less')
end
```

Pascal later made a final ; optional.

Summary

Compiler: scanner, parser, AST, IR, assembly

Scanner divides input into tokens

Scanning defined using a regular language

Parser uses rules to recognize phrases and build AST

Context-free grammars used for parsers

Operator precedence and associativity

Top-down and bottom-up parsers