Compiler Construction
Lent Term 2016

Part I : Lectures 1 – 6 (of 16)

The Front End

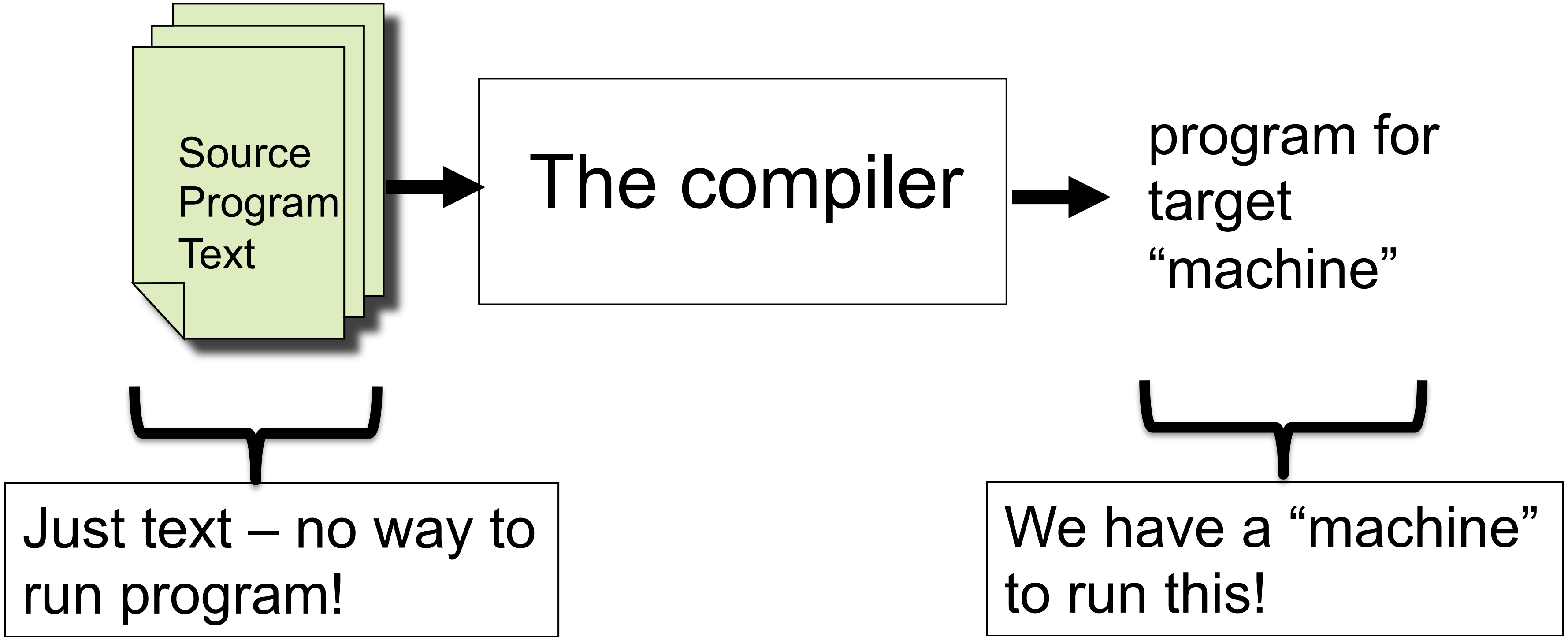
Timothy G. Griffin
tgg22@cam.ac.uk

Computer Laboratory
University of Cambridge

Why Study Compilers?

- Although many of the basic ideas were developed over 50 years ago, compiler construction is still an evolving and active area of research and development.
- Compilers are intimately related to programming language design and evolution.
- Compilers are a Computer Science success story illustrating the hallmarks of our field --- higher-level abstractions implemented with lower-level abstractions.
- Every Computer Scientist should have a basic understanding of how compilers work.

Compilation is a special kind of translation



A good compiler should ...

This course!
- **be correct in the sense that meaning is preserved**
- **produce usable error messages**

OptComp, Part II
- **generate efficient code**
- **itself be efficient**
- **be well-structured and maintainable**

Pick any 2?

Just 1?

Mind The Gap

High Level Language

- “Machine” independent
- Complex syntax
- Complex type system
- Variables
- Nested scope
- Procedures, functions
- Objects
- Modules
- …

Typical Target Language

- “Machine” specific
- Simple syntax
- Simple types
- memory, registers, words
- Single flat scope

Help!!! Where do we begin???

The Gap, illustrated

```
public class Fibonacci {
  public static long fib(int m) {
    if (m == 0) return 1;
    else if (m == 1) return 1;
        else return
                fib(m - 1) + fib(m - 2);
  }
  public static void
    main(String[] args) {
    int m =
        Integer.parseInt(args[0]);
    System.out.println(
      fib(m) + "\n");
  }
}
```

javac Fibonacci.java
javap –c Fibonacci.class

```
public class Fibonacci {
  public Fibonacci();
   Code:
     0: aload_0
     1: invokespecial #1
     4: return
  public static long fib(int);
   Code:
     0: iload_0
     1: ifne        6
     4: lconst_1
     5: lreturn
     6: iload_0
     7: iconst_1
     8: if_icmpne     13
    11: lconst_1
    12: lreturn
    13: iload_0
    14: iconst_1
    15: isub
    16: invokestatic  #2
    19: iload_0
    20: iconst_2
    21: isub
    22: invokestatic  #2
    25: ladd
    26: lreturn
public static void
  main(java.lang.String[]);
 Code:
   0: aload_0
   1: iconst_0
   2: aaload
   3: invokestatic  #3
   6: istore_1
   7: getstatic     #4
  10: new           #5
  13: dup
  14: invokespecial #6
  17: iload_1
  18: invokestatic  #2
  21: invokevirtual #7
  24: ldc           #8
  26: invokevirtual #9
  29: invokevirtual #10
  32: invokevirtual #11
  35: return
}
```

JVM bytecodes

The Gap, illustrated

fib.ml

```
(* fib : int -> int *)
let rec fib m =
   if m = 0
   then 1
   else if m = 1
        then 1
        else fib(m - 1) + fib (m - 2)
```

ocamlc –dinstr fib.ml

```
        branch L2
L1:     acc 0
        push
        const 0
        eqint
        branchifnot L4
        const 1
        return 1
L4:     acc 0
        push
        const 1
        eqint
        branchifnot L3
        const 1
        return 1
L3:     acc 0
        offsetint -2
        push
        offsetclosure 0
        apply 1
        push
        acc 1
        offsetint -1
        push
        offsetclosure 0
        apply 1
        addint
        return 1
L2:     closurerec 1, 0
        acc 0
        makeblock 1, 0
        pop 1
        setglobal Fib!
```

OCaml VM bytecodes

The Gap, illustrated

fib.c

```c
#include<stdio.h>

int Fibonacci(int);
int main()
{
  int n;
  scanf("%d",&n);
  printf("%d\n", Fibonacci(n));
  return 0;
}

int Fibonacci(int n)
{
  if ( n == 0 ) return 0;
  else if ( n == 1 ) return 1;
  else return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

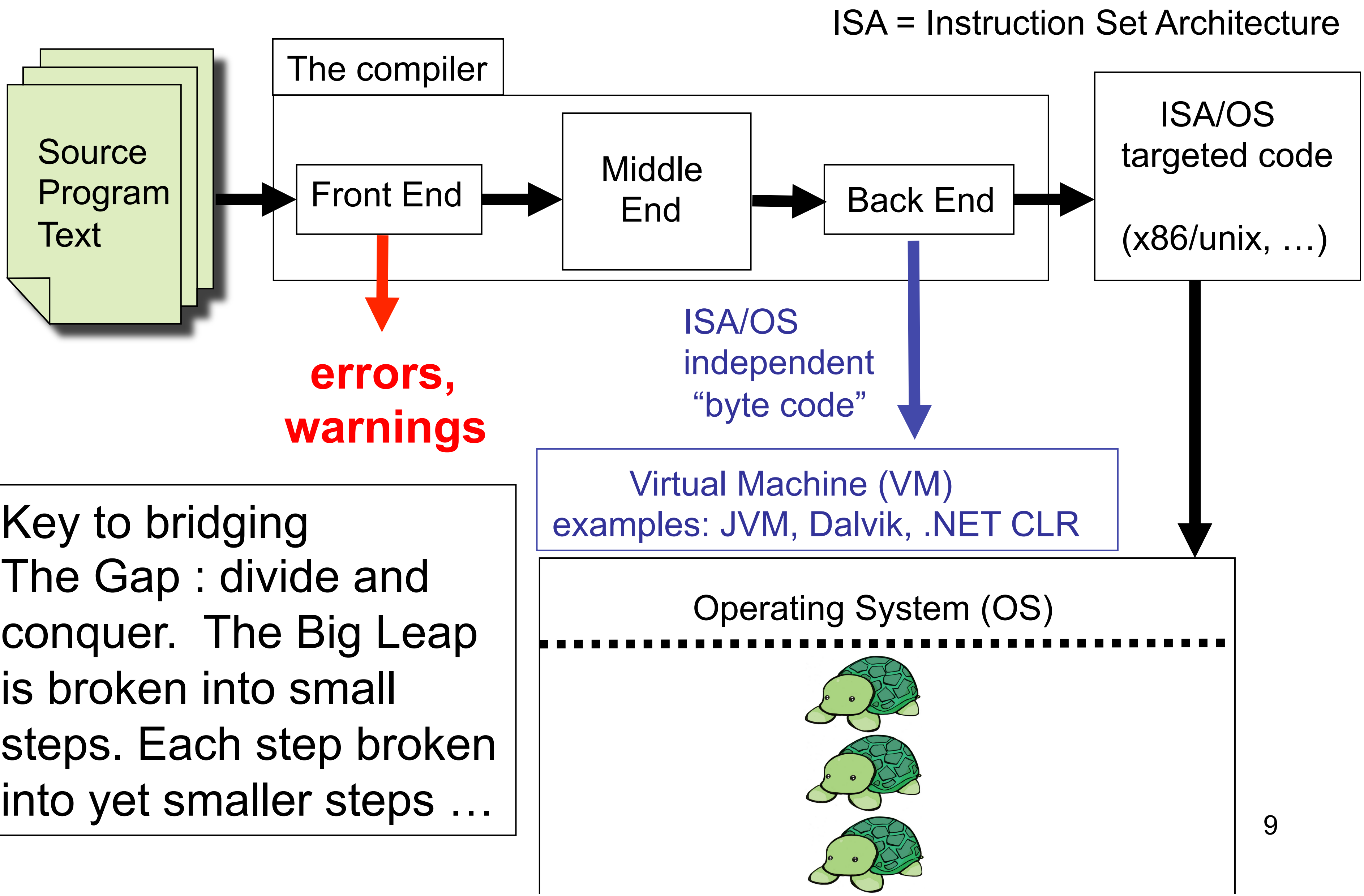
gcc –S fib.c

The Gap, illustrated

```
                .section        __TEXT,__text,regular,pure_instructions
                .globl          _main
                .align          4, 0x90
_main:                  ## @main
                .cfi_startproc
## BB#0:
                pushq           %rbp
Ltmp2:
                .cfi_def_cfa_offset 16
Ltmp3:
                .cfi_offset %rbp, -16
                movq            %rsp, %rbp
Ltmp4:
                .cfi_def_cfa_register %rbp
                subq            $16, %rsp
                leaq            L_.str(%rip), %rdi
                leaq            -8(%rbp), %rsi
                movl            $0, -4(%rbp)
                movb            $0, %al
                callq           _scanf
                movl            -8(%rbp), %edi
                movl            %eax, -12(%rbp)     ## 4-byte Spill
                callq           _Fibonacci
                leaq            L_.str1(%rip), %rdi
                movl            %eax, %esi
                movb            $0, %al
                callq           _printf
                movl            $0, %esi
                movl            %eax, -16(%rbp)     ## 4-byte Spill
                movl            %esi, %eax
                addq            $16, %rsp
                popq            %rbp
                ret
                .cfi_endproc

                .globl          _Fibonacci
                .align          4, 0x90
_Fibonacci:             ## @Fibonacci
                .cfi_startproc
## BB#0:
                pushq           %rbp
Ltmp7:
                .cfi_def_cfa_offset 16
Ltmp8:
                .cfi_offset %rbp, -16
                movq            %rsp, %rbp
Ltmp9:
                .cfi_def_cfa_register %rbp
                subq            $16, %rsp
                movl            %edi, -8(%rbp)
                cmpl            $0, -8(%rbp)
                jne             LBB1_2
## BB#1:
                movl            $0, -4(%rbp)
                jmp             LBB1_5
LBB1_2:
                cmpl            $1, -8(%rbp)
                jne             LBB1_4
## BB#3:
                movl            $1, -4(%rbp)
                jmp             LBB1_5
LBB1_4:
                movl            -8(%rbp), %eax
                subl            $1, %eax
                movl            %eax, %edi
                callq           _Fibonacci
                movl            -8(%rbp), %edi
                subl            $2, %edi
                movl            %eax, -12(%rbp)     ## 4-byte Spill
                callq           _Fibonacci
                movl            -12(%rbp), %edi     ## 4-byte Reload
                addl            %eax, %edi
                movl            %edi, -4(%rbp)
LBB1_5:
                movl            -4(%rbp), %eax
                addq            $16, %rsp
                popq            %rbp
                ret
                .cfi_endproc

                .section        __TEXT,__cstring,cstring_literals
L_.str:                 ## @.str
                .asciz          "%d"

L_.str1:                ## @.str1
                .asciz          "%d\n"


.subsections_via_symbols
```

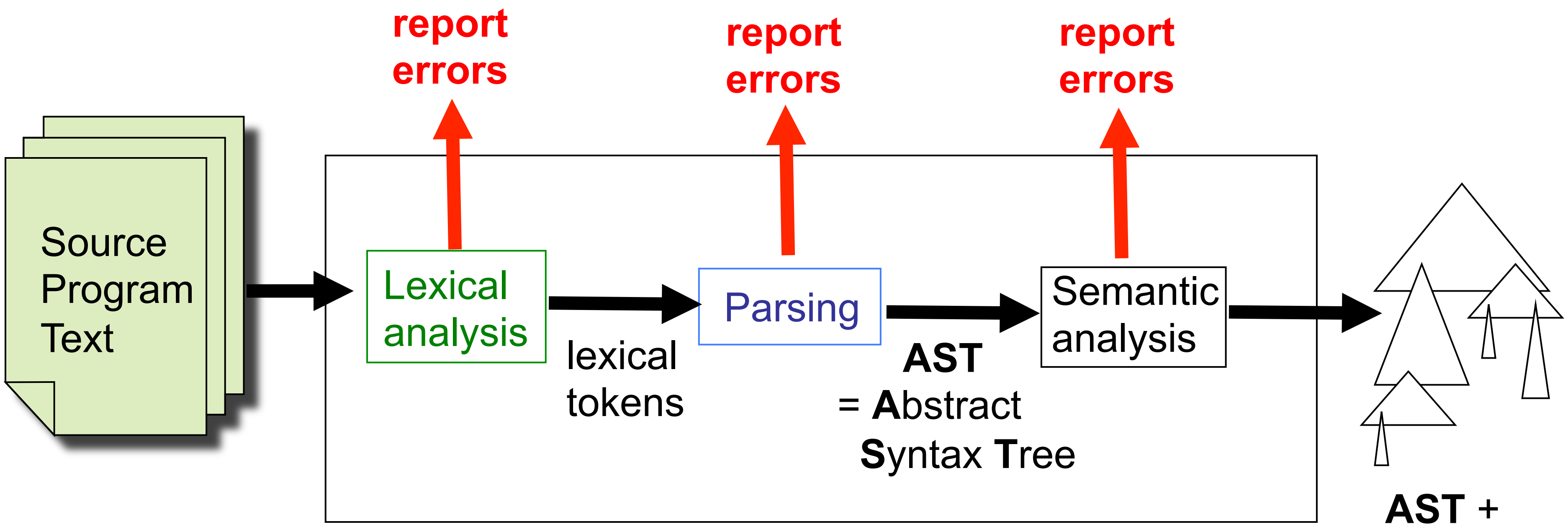
x86/Mac OS

Conceptual view of a typical compiler



Key to bridging The Gap : divide and conquer. The Big Leap is broken into small steps. Each step broken into yet smaller steps …

The shape of a typical “front end”



Lexical theory based on finite automaton and regular expressions

Parsing Theory based on push-down automaton and context-free grammars

Enforce “static sematics” of language: type checking, def/use rules, and so on (**SPL**!)

The AST output from the front-end should represent a <u>legal program</u> in the source language. (“Legal” of course does not mean “bug-free”!)

SPL = Semantics of Programming Languages, Part 1B

Our view of the middle- and back-ends : a sequence of small transformations

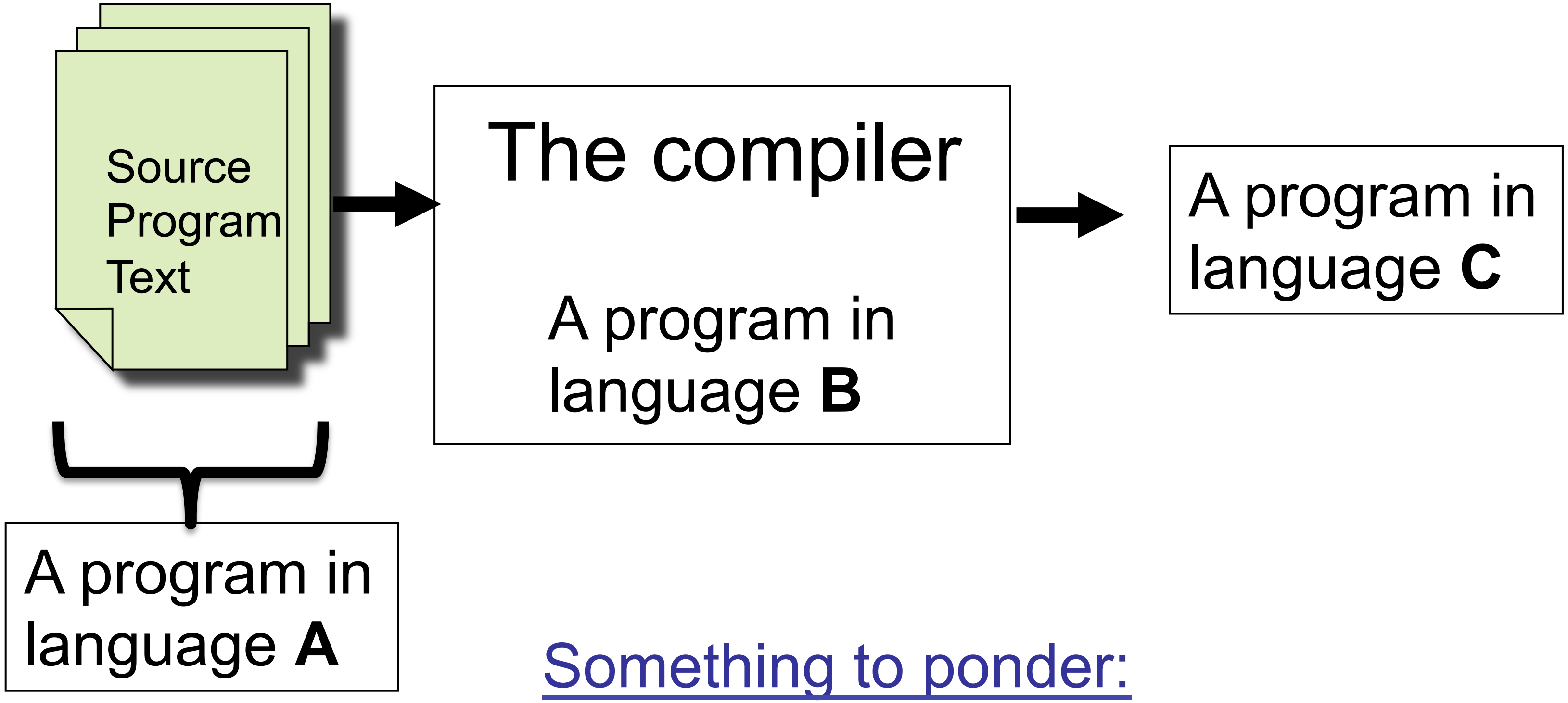
Intermediate Languages

$$\overbrace{\text{IL-1} \longrightarrow \text{IL-2} \longrightarrow \cdots \longrightarrow \text{IL-k}}$$

Of course industrial-strength compilers may collapse many small-steps …

- Each **IL** has its own semantics (perhaps informal)
- Each transformation ($\longrightarrow$) preserves semantics (**SPL**!)
- Each transformation eliminates only a few aspects of **the gap**
- Each transformation is fairly easy to understand
- Some transformations can be described as “optimizations”
- We will associate each **IL** with its own interpreter/VM. (Again, not something typically done in “industrial-strength” compilers.)

Compilers must be compiled



<u>Something to ponder:</u>
A compiler is just a program.
But how did it get compiled?
The OCaml compiler is written in OCaml.

How was the compiler compiled?

Approach Taken

- We will develop a compiler for a fragment of L3 introduced in Semantics of Programming Languages, Part 1B.
- We will pay special attention to the correctness.
- We will compile only to Virtual Machines (VMs) of various kinds. See Part II optimising compilers for generating lower-level code.
- Our toy compiler is available on the course web site.
- We will be using the **OCaml** dialect of ML.

- Install from https://ocaml.org.
- See OCaml Labs : http://www.cl.cam.ac.uk/projects/ocamllabs.
- A side-by-side comparison of SML and OCaml Syntax: http://www.mpi-sws.org/~rossberg/sml-vs-ocaml.html

SML Syntax vs. OCaml Syntax

```
datatype 'a tree =
   Leaf of 'a
  | Node of 'a * ('a tree) * ('a tree)

fun map_tree f (Leaf a) = Leaf (f a)
    | map_tree f (Node (a, left, right)) =
       Node(f a, map_tree f left, map_tree f right)


let val l =
     map_tree (fn a => [a]) [Leaf 17, Leaf 21]
in
     List.rev l
end
```

```
type 'a tree =
   Leaf of 'a
  | Node of 'a * ('a tree) * ('a tree)

let rec map_tree f = function
  | Leaf a -> Leaf (f a)
  | Node (a, left, right)  ->
     Node(f a, map_tree f left, map_tree f right)

let l =
    map_tree (fun a -> [a]) [Leaf 17; Leaf 21]
in
    List.rev l
```

The Shape of this Course

1. Overview
2. Slang Front-end, Slang demo. Code tour.
3. Lexical analysis : application of Theory of Regular Languages and Finite Automata
4. Generating Recursive descent parsers
5. Beyond Recursive Descent Parsing I
6. Beyond Recursive Descent Parsing II
7. High-level "definitional" interpreter (interpreter 0). Make the stack explicit and derive interpreter 2
8. Flatten code into linear array, derive interpreter 3
9. Move complex data from stack into the heap, derive the Jargon Virtual Machine (interpreter 4)
10. More on Jargon VM. Environment management. Static links on stack. Closures.
11. A few program transformations. Tail Recursion Elimination (TRE), Continuation Passing Style (CPS). Defunctionalisation (DFC)
12. CPS+TRE+DFC provides a formal way of understanding how we went from interpreter 0 to interpreter 2. We fill the gap with interpreter 1
13. Bootstrapping a compiler
14. Run-time environments, automated memory management ("garbage collection")
15. Assorted topics : exceptions, objects, compilation units, linking
16. Assorted topics : simple optimisations, stack machine vs. register

LECTURE 2
Slang Front End

- Slang (= Simple LANGuage)
  - A subset of L3 from Semantics …
  - … with very ugly concrete syntax
  - You are invited to experiment with improvements to this concrete syntax.
- Slang : concrete syntax, types
- Abstract Syntax Trees (ASTs)
- The Front End
- A short in-lecture demo of slang and a brief tour of the code …

Clunky Slang Syntax (informal)

```
uop := - | ~

bop ::= + | - | * | < | = | && | ||

t ::= bool | int | unit | (t) | t * t | t + t | t -> t | t ref

e ::= () | n | true | false | x | (e) | ? |
      e bop e |  uop e |
      if e then else e end |
      e e | fun (x : t) -> e end |
      let x : t = e in e end |
      let f(x : t) : t = e in e end |
      !e | ref e | e := e | while e do e end |
      begin e; e; … e end |
      (e, e) | snd e | fst e |
      inl t e | inr t e |
      case e of inl(x : t) -> e | inr(x:t) -> e end
```

(~ is boolean negation)

(? requests an integer
input from terminal)

(notice type annotation
on inl and inr constructs)

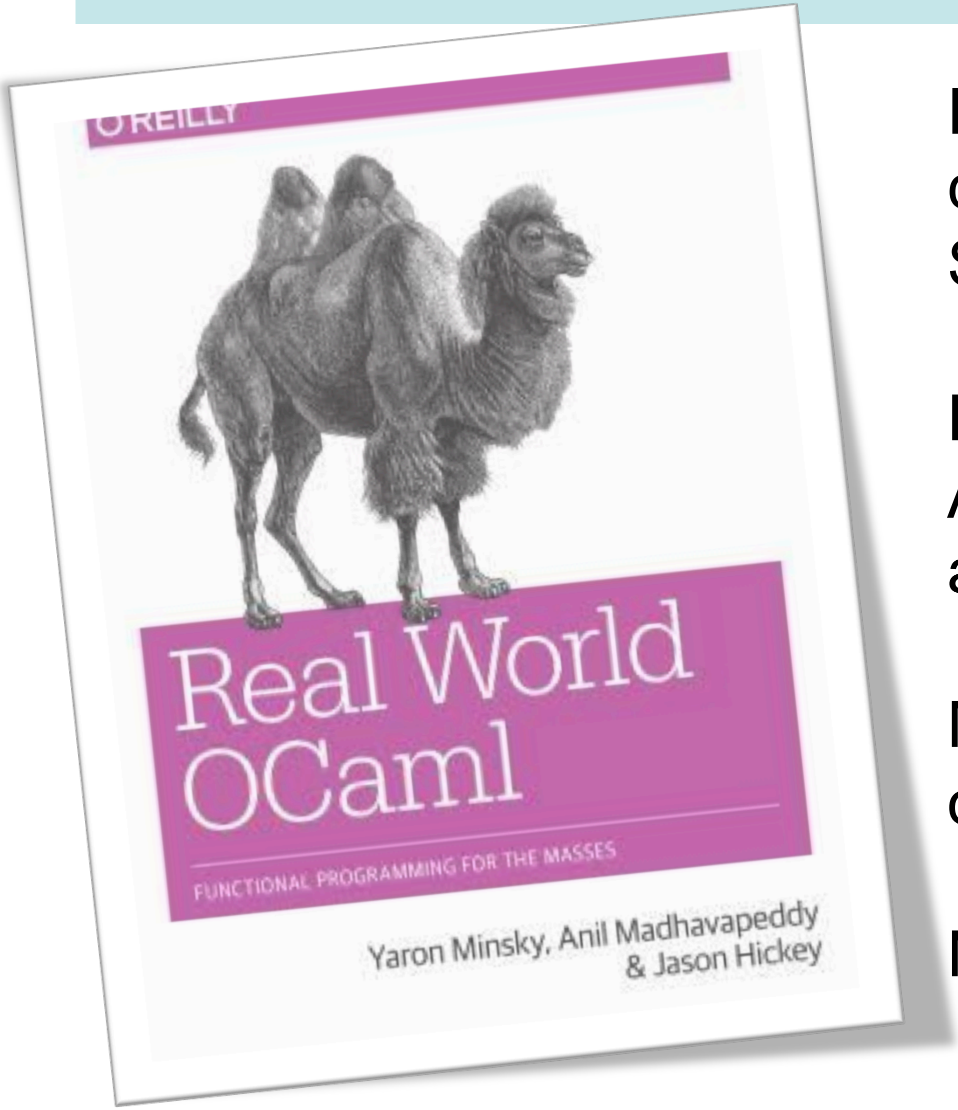
From slang/examples

```
let fib( m : int) : int =
   if m = 0
   then 1
   else if m = 1
         then 1
          else fib (m - 1) +
                fib (m -2)
          end
    end
in
   fib(?)
end
```

```
let gcd( p : int * int) : int =
   let m : int = fst p
   in let  n : int = snd p
   in  if m = n
       then m
       else if m < n
             then gcd(m, n - m)
             else  gcd(m - n, n)
             end
        end
      end
    end
in gcd(?, ?) end
```

The ? requests an integer input from the terminal

CONTEST!



For the most elegant concrete syntax for the Slang fragment of L3.

Reduce required keyword usage AND make some of the type annotations optional.

Must be in OCaml. Must use ocamlyacc.

No parser conflicts allowed!

WIN A COPY!

Slang Front End

Input file foo.slang

Parse (we use Ocaml versions of LEX and YACC, covered in Lectures 3 --- 6)

Parsed AST (Past.expr)

Static analysis : check types, and context-sensitive rules, resolve overloaded operators

Parsed AST (Past.expr)

Remove "syntactic sugar", file location information, and most type information

Intermediate AST (Ast.expr)

Parsed AST (past.ml)

```
type var = string

type loc = Lexing.position

type type_expr =
   | TEint
   | TEbool
   | TEunit
   | TEref of type_expr
   | TEarrow of type_expr * type_expr
   | TEproduct of type_expr * type_expr
   | TEunion of type_expr * type_expr

type oper = ADD | MUL | SUB | LT |
            AND | OR | EQ | EQB | EQI

type unary_oper = NEG | NOT
```

Locations (loc) are used in generating error messages.

```
type expr =
     | Unit of loc
     | What of loc
     | Var of loc * var
     | Integer of loc * int
     | Boolean of loc * bool
     | UnaryOp of loc * unary_oper * expr
     | Op of loc * expr * oper * expr
     | If of loc * expr * expr * expr
     | Pair of loc * expr * expr
     | Fst of loc * expr
     | Snd of loc * expr
     | Inl of loc * type_expr * expr
     | Inr of loc * type_expr * expr
     | Case of loc * expr * lambda * lambda
     | While of loc * expr * expr
     | Seq of loc * (expr list)
     | Ref of loc * expr
     | Deref of loc * expr
     | Assign of loc * expr * expr
     | Lambda of loc * lambda
     | App of loc * expr * expr
     | Let of loc * var * type_expr * expr * expr
     | LetFun of loc * var * lambda
                   * type_expr * expr
     | LetRecFun of loc * var * lambda
                   * type_expr * expr
```

static.mli, static.ml

```
val infer : (Past.var * Past.type_expr) list -> (Past.expr * Past.type_expr)

val check : Past.expr -> Past.expr   (* infer on empty environment *)
```

- Check type correctness
- Rewrite expressions to resolve EQ to EQI (for integers) or EQB (for bools).
- Only LetFun is returned by parser. Rewrite to LetRecFun when function is actually recursive.

Lesson : while enforcing “context-sensitive rules” we can resolve ambiguities that cannot be specified in context-free grammars.

Internal AST (ast.ml)

```
type var = string

type oper = ADD | MUL | SUB | LT |
            AND | OR | EQB | EQI

type unary_oper = NEG | NOT | READ
```

No locations, types.
No Let, EQ.

Is getting rid of types a bad idea? Perhaps a full answer would be language-dependent…

```
type expr =
      | Unit
      | Var of var
      | Integer of int
      | Boolean of bool
      | UnaryOp of unary_oper * expr
      | Op of expr * oper * expr
      | If of expr * expr * expr
      | Pair of expr * expr
      | Fst of expr
      | Snd of expr
      | Inl of expr
      | Inr of expr
      | Case of expr * lambda * lambda
      | While of expr * expr
      | Seq of (expr list)
      | Ref of expr
      | Deref of expr
      | Assign of expr * expr
      | Lambda of lambda
      | App of expr * expr
      | LetFun of var * lambda * expr
      | LetRecFun of var * lambda * expr

and lambda = var * expr
```

past_to_ast.ml

```
val translate_expr : Past.expr -> Ast.expr
```

```
let x : t  = e1 in e2 end
```

```
(fun (x: t) -> e2 end) e1
```

This is done to simplify some of our code.
Is it a good idea?   Perhaps not.

Lecture 3, 4, 5, 6
Lexical Analysis and Parsing

1. Theory of Regular Languages and Finite Automata applied to lexical analysis.
2. Context-free grammars
3. The ambiguity problem
4. Generating Recursive descent parsers
5. Beyond Recursive Descent Parsing I
6. Beyond Recursive Descent Parsing II

What problem are we solving?

Translate a sequence of characters

```
if m = 0 then 1 else if m = 1 then 1 else fib (m - 1) +  fib (m -2)
```

into a sequence of **tokens**

```
IF, IDENT "m", EQUAL, INT 0, THEN, INT 1, ELSE, IF,
IDENT "m", EQUAL, INT 1, THEN, INT 1, ELSE, IDENT "fib",
LPAREN, IDENT "m", SUB, INT 1, RPAREN, ADD,
IDENT "fib", LPAREN, IDENT "m", SUB, INT 2, RPAREN
```

implemented with some data type

```
type token =
    | INT of int| IDENT of string | LPAREN | RPAREN
    | ADD | SUB | EQUAL | IF | THEN | ELSE
    | ...
```

Recall from Discrete Mathematics (Part 1A)

Regular expressions (concrete syntax)

over a given alphabet Σ. $\{\varepsilon, \emptyset, |, *, (,)\}$

Let Σ' be the ~~4~~ 6-element set $\cancel{\{\epsilon, \emptyset, |, *\}}$ (assumed disjoint from Σ)

$$U = (\Sigma \cup \Sigma')^*$$

axioms: $\dfrac{}{a}$ $\quad \dfrac{}{\epsilon}$ $\quad \dfrac{}{\emptyset}$

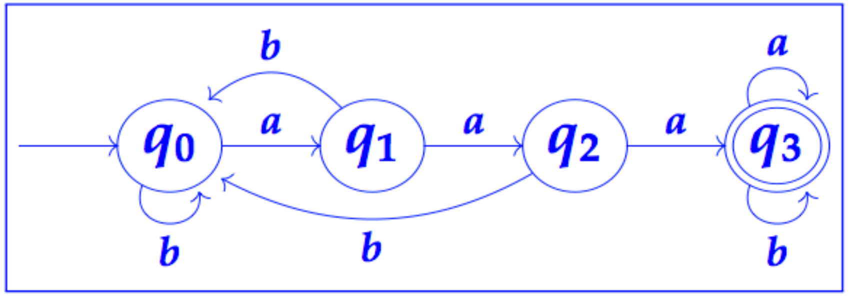
rules: $\dfrac{r}{(r)}$ $\quad \dfrac{r \quad s}{r|s}$ $\quad \dfrac{r \quad s}{rs}$ $\quad \dfrac{r}{r^*}$

(where $a \in \Sigma$ and $r, s \in U$)

Recall from Discrete Mathematics (Part 1A)

Example of a finite automaton

$M \triangleq$



- set of states: $\{q_0, q_1, q_2, q_3\}$
- input alphabet: $\{a, b\}$
- transitions, labelled by input symbols: as indicated by the above directed graph
- start state: q_0
- accepting state(s): q_3

Recall from Discrete Mathematics (Part 1A)

Kleene's Theorem

Definition. A language is **regular** iff it is equal to $L(M)$, the set of strings accepted by some deterministic finite automaton M.

Theorem.

(a) For any regular expression r, the set $L(r)$ of strings matching r is a regular language.

(b) Conversely, every regular language is the form $L(r)$ for some regular expression r.

Traditional Regular Language Problem

Given a regular expression,

$$e$$

and an input string w; determine if $w \in L(e)$

.

Construct a DFA M from *e* and test if it accepts *w*.

Recall construction : regular expression → NFA → DFA

Something closer to the "lexing problem"

Given an ordered list of regular expressions,

$$e_1 \quad e_2 \quad \cdots \quad e_k$$

and an input string w, find a list of pairs

$$(i_1, w_1), (i_2, w_2), \ldots (i_n, w_n)$$

such that

$$1) \quad w = w_1 w_2 \ldots w_n$$

$$2) \quad w_j \in L(e_{i_j})$$

$$3) \quad w_j \in L(e_s) \rightarrow i_j \leq s \quad \text{(priority rule)}$$

.

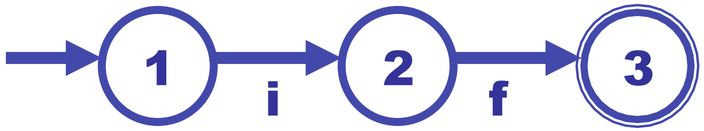
$$4) \quad \forall j : \forall u \in \text{prefix}(w_{j+1} w_{j+2} \cdots w_n) : u \neq \varepsilon$$
$$\rightarrow \forall s : w_j u \notin L(e_s) \quad \text{(longest match)}$$

Why ordered? Is "if" a variable or a keyword? Need priority to resolve ambiguity.

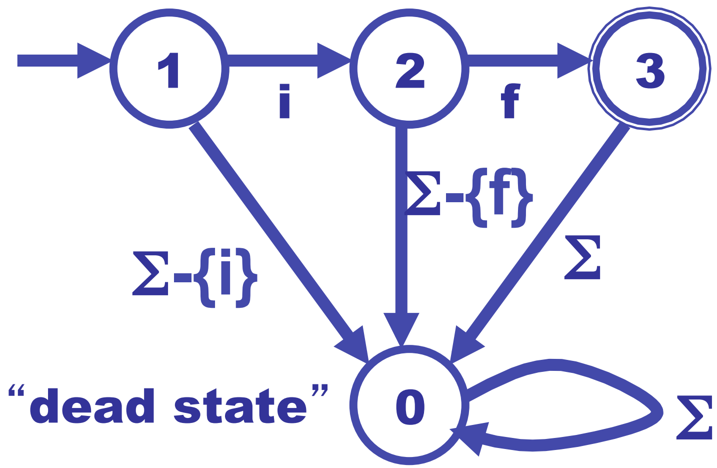
Why longest match? Is "ifif" a variable or two "if" keywords?

Define Tokens with Regular Expressions (Finite Automata)

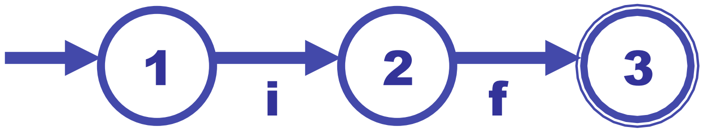
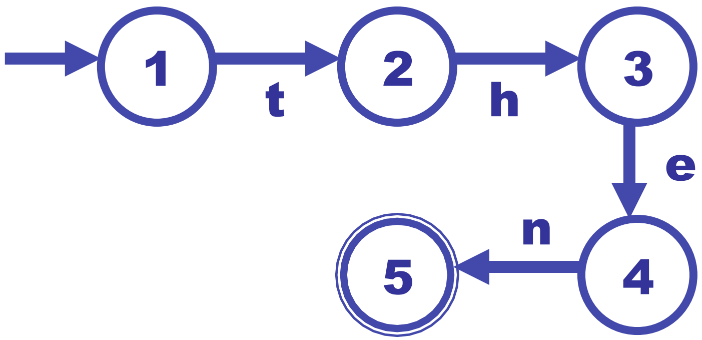
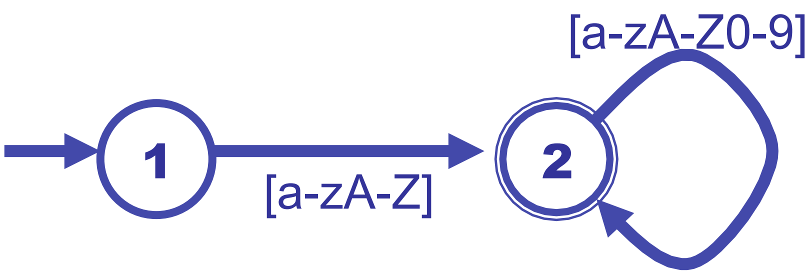
Keyword: if



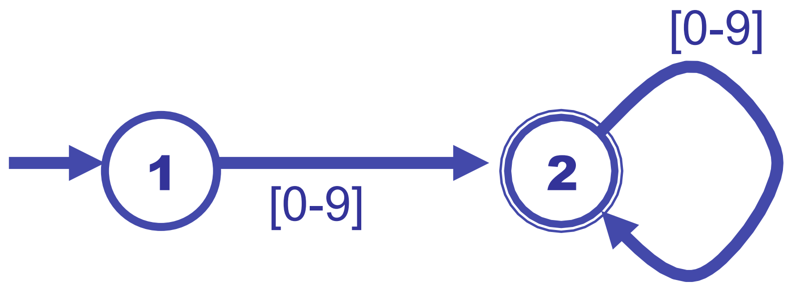
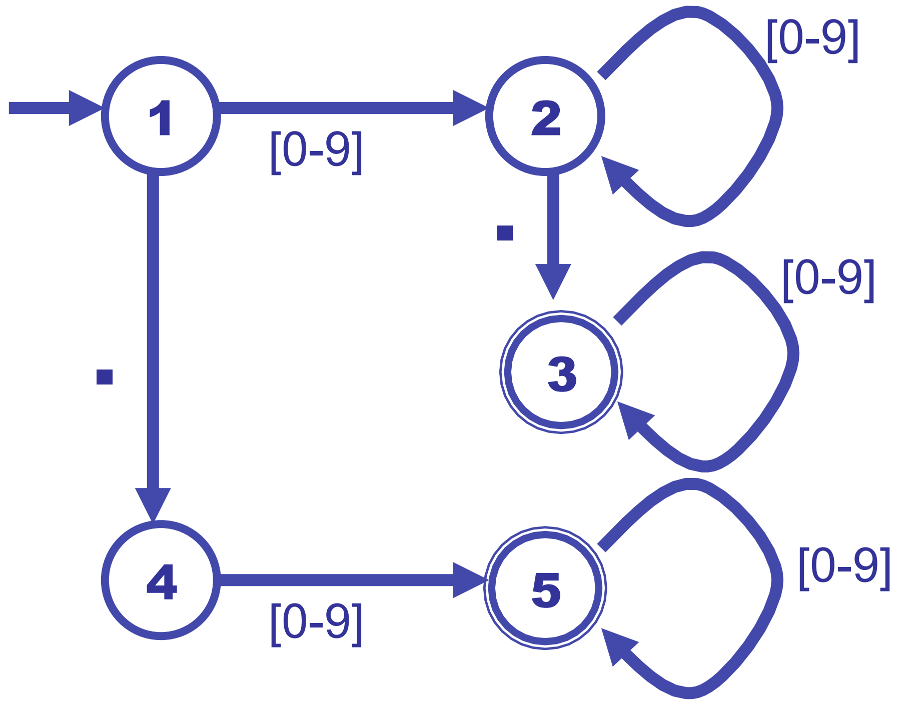
This FA is really shorthand for:



Define Tokens with Regular Expressions (Finite Automata)

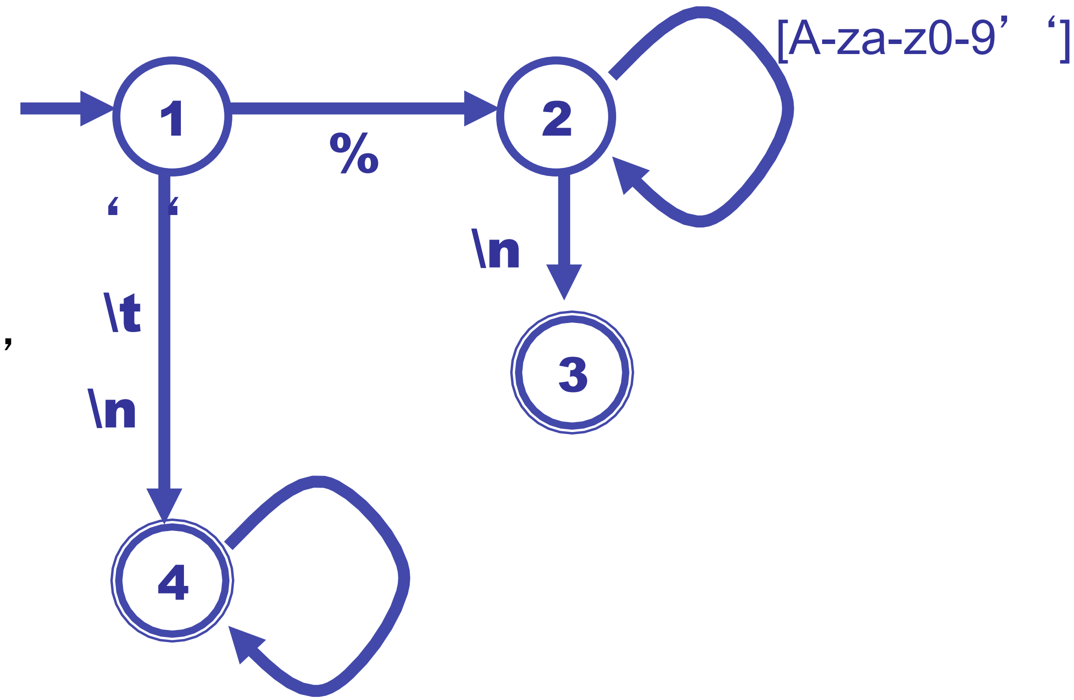
Regular Expression	Finite Automata	Token
Keyword: if		**KEY(IF)**
Keyword: then		**KEY(then)**
Identifier: [a-zA-Z][a-zA-Z0-9]*		**ID(s)**

Define Tokens with Regular Expressions (Finite Automata)

Regular Expression	Finite Automata	Token
number: **[0-9][0-9]***		**NUM(n)**
real: **([0-9]+ ‘.’ [0-9]*) | ([0-9]* ‘.’ [0-9]+)**		**NUM(n)**

No Tokens for "White-Space"

White-space:
(' ' | '\n' | '\t')+
| '%' [A-Za-z0-9' ']+' \n'



Constructing a Lexer

INPUT: **an <u>ordered</u> list of regular expressions**

e_1
e_2
$\vdots$
e_k

Construct all corresponding finite automata

NFA_1
NFA_2
$\vdots$
NFA_k

use priority

Construct a single non-deterministic finite automata

NFA

Construct a single deterministic finite automata

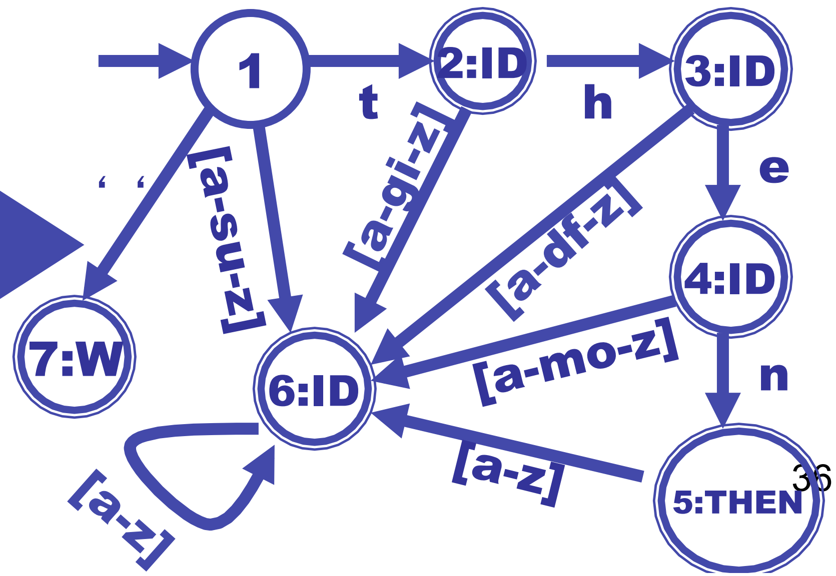
DFA

---

(1) Keyword : then

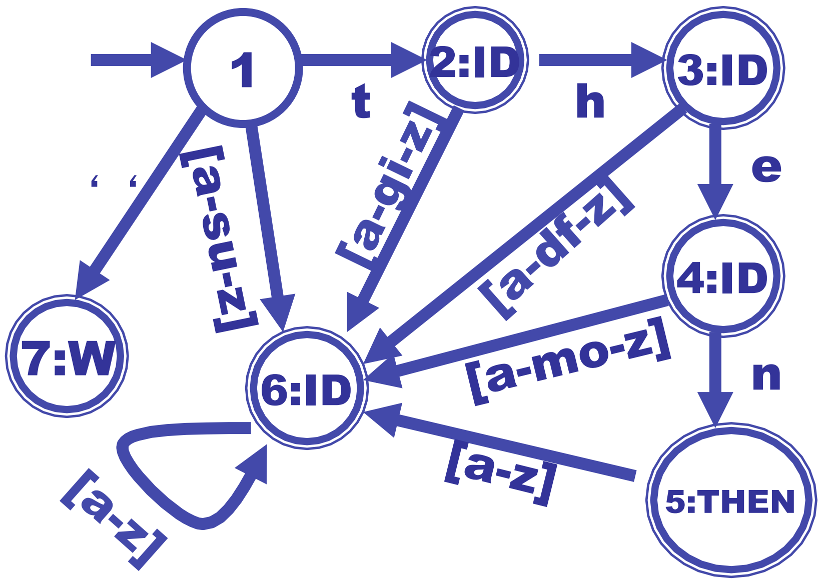
(2) Ident : [a-z][a-z]*

(2) White-space: ‘ ‘



What about longest match?

Start in initial state,
Repeat:
(1) read input until dead state is reached. Emit token associated with last accepting state.
(2) reset state to start state

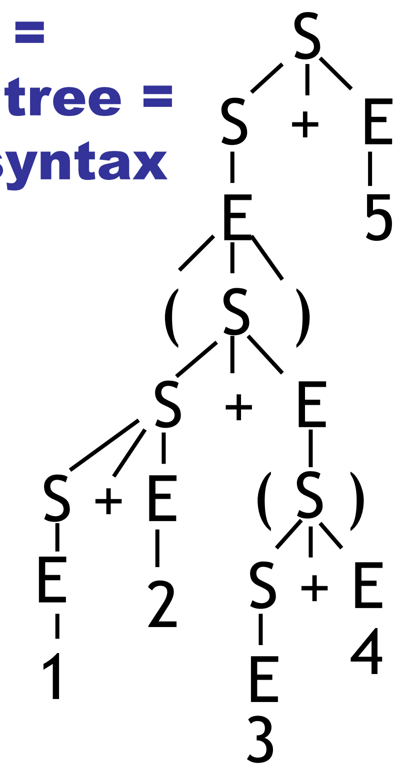


| = current position, $ = EOF

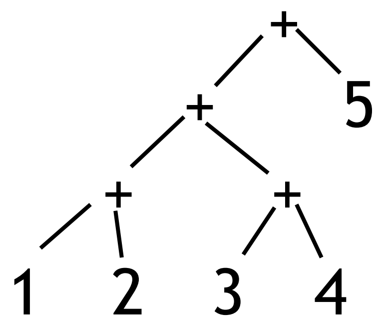
<table>
<tr><th>Input</th><th>current state</th><th>last accepting state</th></tr>
<tr><td>|then thenx$</td><td>1</td><td>0</td></tr>
<tr><td>t|hen thenx$</td><td>2</td><td>2</td></tr>
<tr><td>th|en thenx$</td><td>3</td><td>3</td></tr>
<tr><td>the|n thenx$</td><td>4</td><td>4</td></tr>
<tr><td>then| thenx$</td><td>5</td><td>5</td></tr>
<tr><td>then |thenx$</td><td>0</td><td>5 EMIT KEY(THEN)</td></tr>
<tr><td>then| thenx$</td><td>1</td><td>0 RESET</td></tr>
<tr><td>then |thenx$</td><td>7</td><td>7</td></tr>
<tr><td>then t|henx$</td><td>0</td><td>7 EMIT WHITE(' ')</td></tr>
<tr><td>then |thenx$</td><td>1</td><td>0 RESET</td></tr>
<tr><td>then t|henx$</td><td>2</td><td>2</td></tr>
<tr><td>then th|enx$</td><td>3</td><td>3</td></tr>
<tr><td>then the|nx$</td><td>4</td><td>4</td></tr>
<tr><td>then then|x$</td><td>5</td><td>5</td></tr>
<tr><td>then thenx|$</td><td>6</td><td>6</td></tr>
<tr><td>then thenx$|</td><td>0</td><td>6 EMIT ID(thenx)</td></tr>
</table>

Concrete vs. Abstract Syntax Trees

parse tree = derivation tree = concrete syntax tree



Abstract Syntax Tree (AST)



An AST contains only the information needed to generate an intermediate representation

Normally a compiler constructs the concrete syntax tree only implicitly (in the parsing process) and explicitly constructs an AST.

On to Context Free Grammars (CFGs)

E ::= ID

E ::= NUM

E ::= E * E

E ::= E / E

E ::= E + E

E ::= E – E

E ::= (E)

E is a *non-terminal symbol*

ID and NUM are *lexical classes*

*, (,), +, and – are *terminal symbols*.

E ::= E + E is called a *production rule*.

Usually will write this way

E ::= ID | NUM | E * E | E / E | E + E | E – E | (E)

CFG Derivations

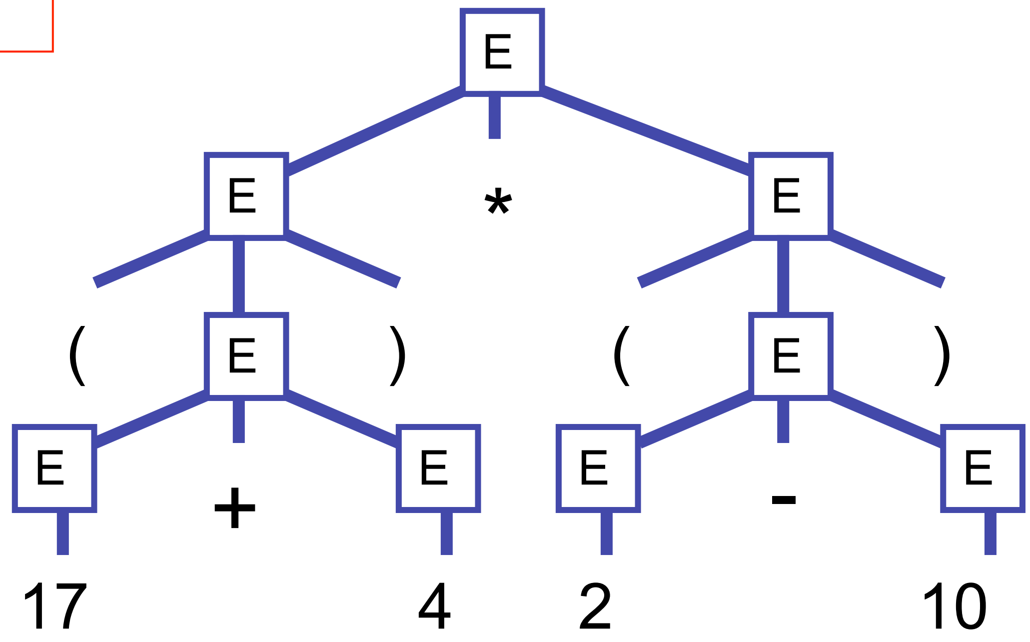
(G1) E ::= ID | NUM | ID | E * E | E / E | E + E | E – E | (E)

E → E * E
→ E * (E)
→ E * (E – E)
→ E * (E – 10)
→ E * (2 – 10)
→ (E) * (2 – 10)
→ (E + E) * (2 – 10)
→ (E + 4) * (2 – E)
→ (17 + 4) * (2 – 10)

Rightmost derivation

E → E * E
→ (**E**) * E
→ (**E** + E) * E
→ (17 + E) * E
→ (17 + 4) * E
→ (17 + 4) * (E)
→ (17 + 4) * (**E** – E)
→ (17 + 4) * (2 – **E**)
→ (17 + 4) * (2 – 10)

Leftmost derivation



The Derivation Tree for
(17 + 4) * (2 – 10)

More formally, ...

- **A CFG is a quadruple G = (N, T, R, S) where**
  - **N is the set of *non-terminal symbols***
  - **T is the set of *terminal symbols* (N and T disjoint)**
  - **$S \in N$ is the *start symbol***
  - **$R \subseteq N \times (N \cup T)^*$ is a set of rules**
- **Example: The grammar of nested parentheses G = (N, T, R, S) where**
  - **N = {S}**
  - **T ={ (,) }**
  - **R ={ (S, (S)) , (S, SS), (S,) }**

We will normally write R as

S ::= (S) | SS |

Derivations, more formally…

- Start from start symbol (*S*)
- Productions are used to derive a sequence of tokens from the start symbol
- For arbitrary strings α, β and γ comprised of both terminal and non-terminal symbols,
  and a production $A \rightarrow \beta$,
  a single step of derivation is
  $\alpha A\gamma \Rightarrow \alpha\beta\gamma$
  - *i.e.*, substitute β for an occurrence of A
- $\alpha \Rightarrow^* \beta$ means that b can be derived from a in 0 or more single steps
- $\alpha \Rightarrow^+ \beta$ means that b can be derived from a in 1 or more single steps

L(G) = The Language Generated by Grammar G

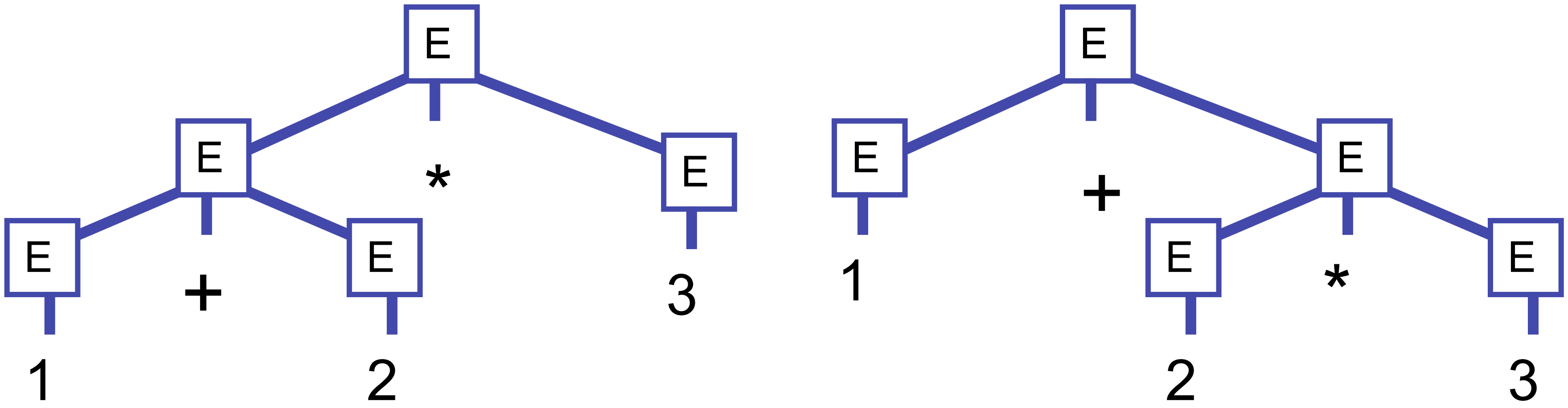
The language generated by G is the set of all terminal strings derivable from the start symbol S:

$$L(G) = \{w \in T^* \mid S \Rightarrow +w\}$$

For any subset W of T*, if there exists a CFG G such that L(G) = W, then W is called a Context-Free Language (CFL) over T.

Ambiguity

(G1) E ::= ID | NUM | ID | E * E | E / E | E + E | E – E | (E)



Both derivation trees correspond to the string

1 + 2 * 3

This type of ambiguity will cause problems when we try to go from strings to derivation trees!

Problem: Generation vs. Parsing

- **Context-Free Grammars (CFGs) describe how to to *generate***
- ***Parsing* is the inverse of generation,**
  - **Given an input string, is it in the language generated by a CFG?**
  - **If so, construct a derivation tree (normally called a *parse tree*).**
  - **Ambiguity is a big problem**

Note : recent work on Parsing Expression Grammars (PEGs) represents an attempt to develop a formalism that describes parsing directly. This is beyond the scope of these lectures …

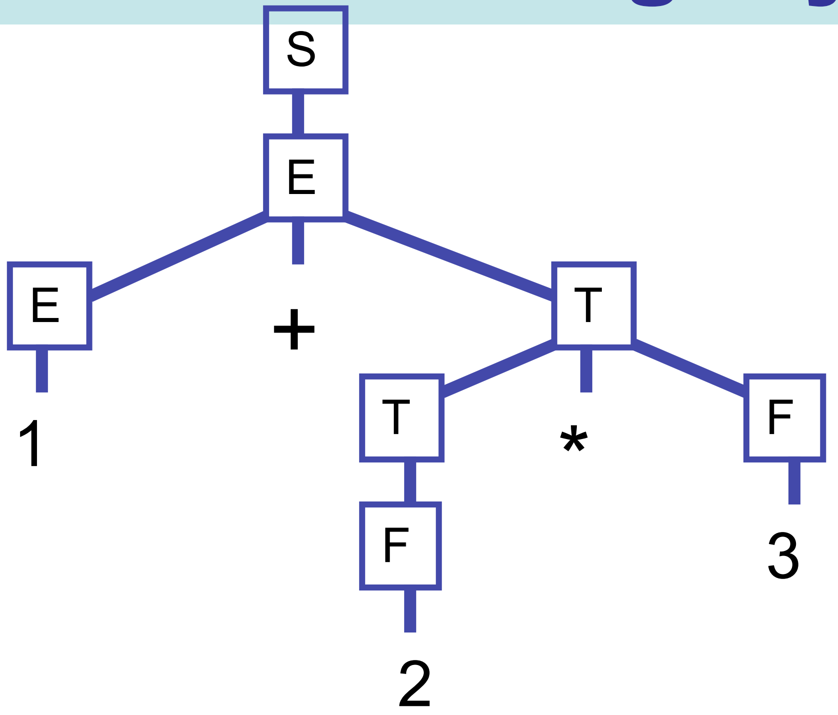
We can often modify the grammar in order to eliminate ambiguity

(G2)
S :: = E$ (start, $ = EOF)

E ::= E + T (expressions)
| E – T
| T

T ::= T * F (terms)
| T / F
| F

F ::= NUM (factors)
| ID
| (E)



This is the *unique* derivation tree for the string

1 + 2 * 3$

Note: L(G1) = L(G2).
Can you prove it?

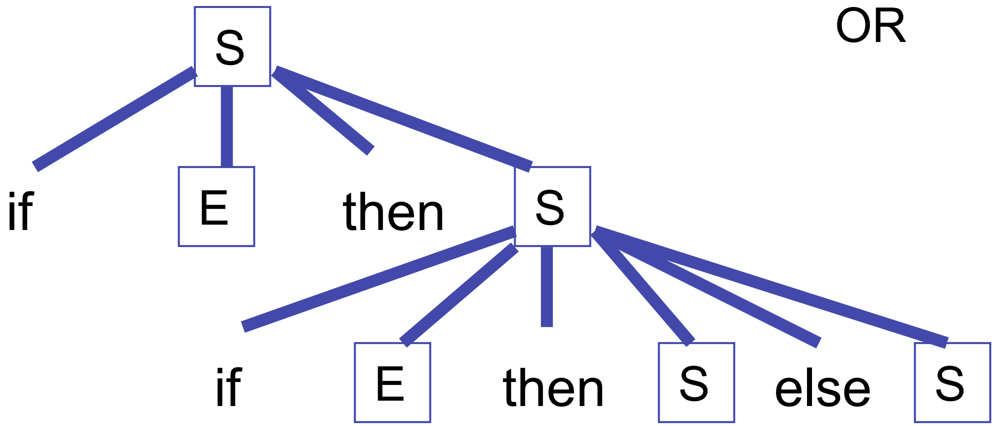
Famously Ambiguous

(G3) S ::= if E then S else S | if E then S | blah-blah

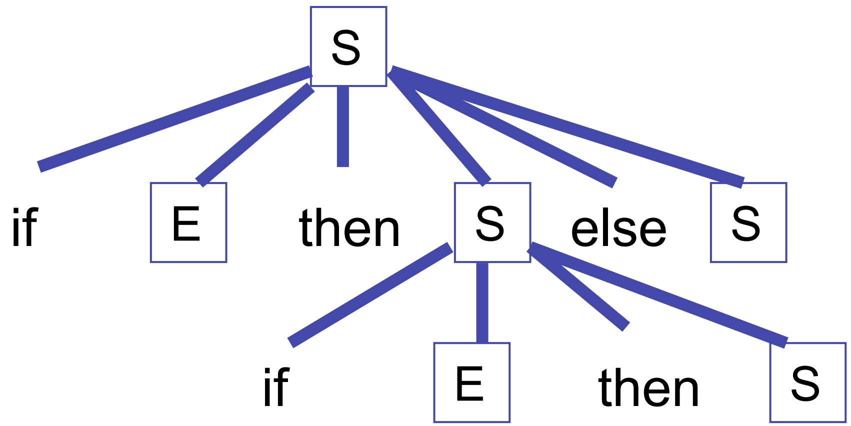
What does

if e1 then if e2 then s1 else s3

mean?



OR



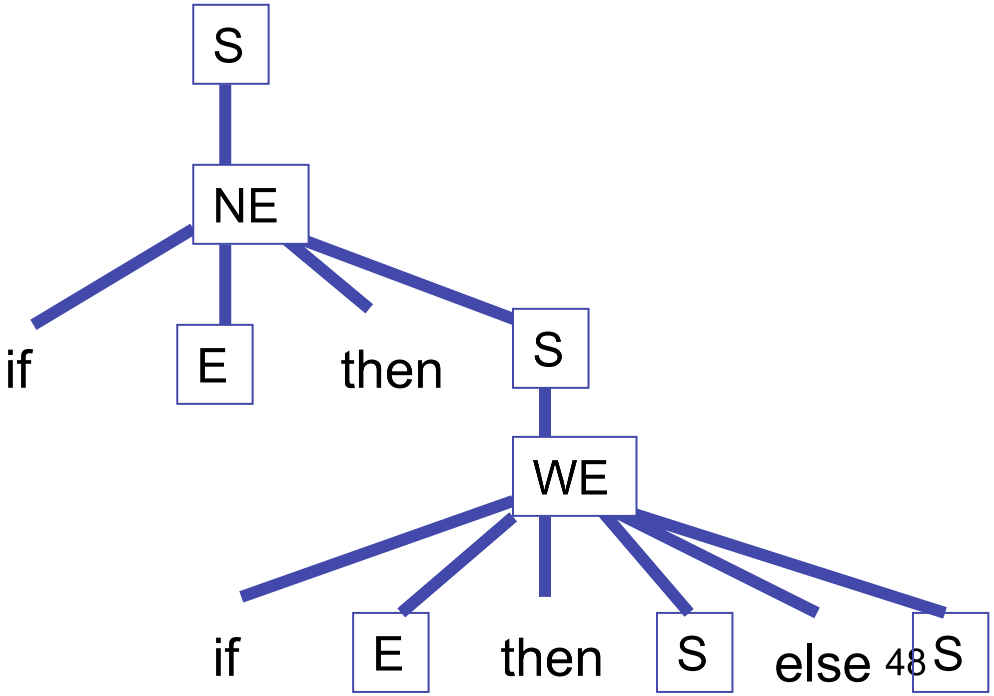
Rewrite?

(G4)
S ::= WE | NE
WE ::= **if** E **then** WE **else** WE | **blah-blah**
NE ::= **if** E **then** S
| **if** E **then** WE **else** NE

Now,

if e1 then if e2 then s1 else s3

has a unique derivation.



Note: L(G3) = L(G4).
Can you prove it?

Fun Fun Facts

See Hopcroft and Ullman, "Introduction to Automata Theory, Languages, and Computation"

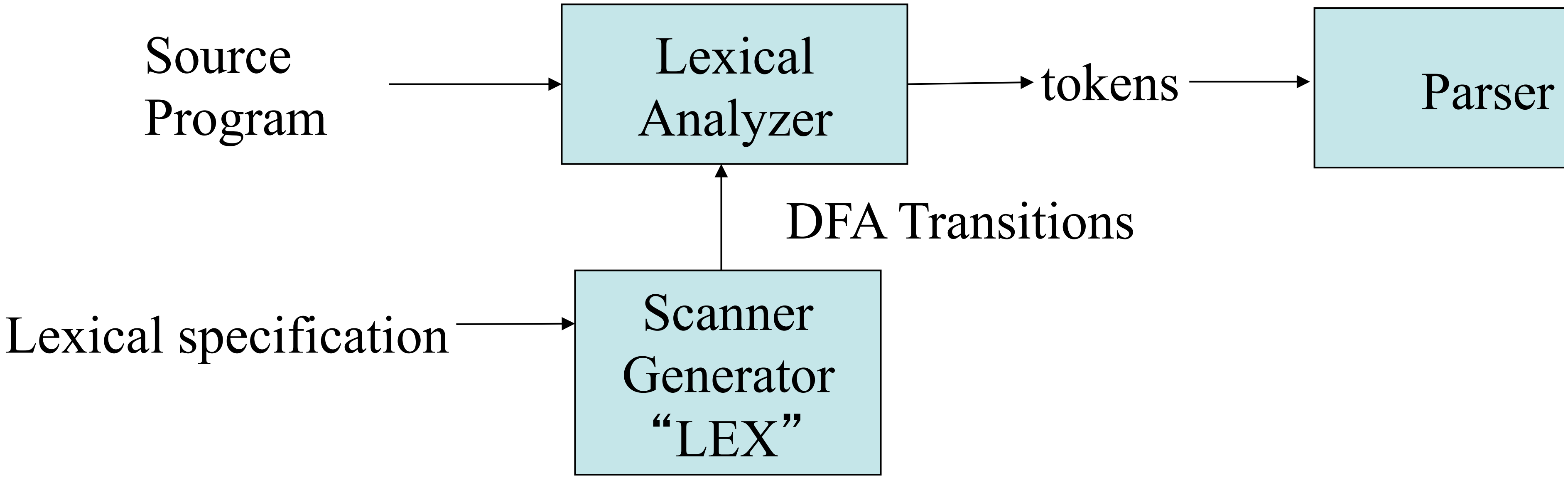
(1) Some context free languages are *inherently ambiguous* --- every context-free grammar will be ambiguous. For example:

$$L = \{a^n b^n c^m d^m \mid m \geq 1, n \geq 1\} \cup \{a^n b^m c^m d^n \mid m \geq 1, n \geq 1\}$$

(2) Checking for ambiguity in an arbitrary context-free grammar is not decidable! Ouch!

(3) Given two grammars G1 and G2, checking L(G1) = L(G2) is not decidable! Ouch!

Generating Lexical Analyzers



The idea : use regular expressions as the basis of a lexical specification. The core of the lexical analyzer is then a deterministic finite automata (DFA)

Predictive (Recursive Descent) Parsing
Can we automate this?

(G5)

S :: = if E then S else S
  | begin S L
  | print E

E ::= NUM = NUM

L ::= end
  | ; S L

```
int tok = getToken();

void advance() {tok = getToken();}
void eat (int t) {if (tok == t) advance(); else error();}

void S() {switch(tok) {
      case IF:    eat(IF); E(); eat(THEN);
                  S(); eat(ELSE); S(); break;
      case BEGIN: eat(BEGIN); S(); L(); break;
      case PRINT: eat(PRINT); E(); break;
      default: error();
     }}

void L() {switch(tok) {
      case END:  eat(END); break;
      case SEMI: eat(SEMI); S(); L(); break;
      default: error();
     }}

void E() {eat(NUM) ; eat(EQ); eat(NUM); }
```

Parse corresponds to a left-most derivation constructed in a “top-down” manner

From Andrew Appel, “Modern Compiler Implementation in Java” page 46

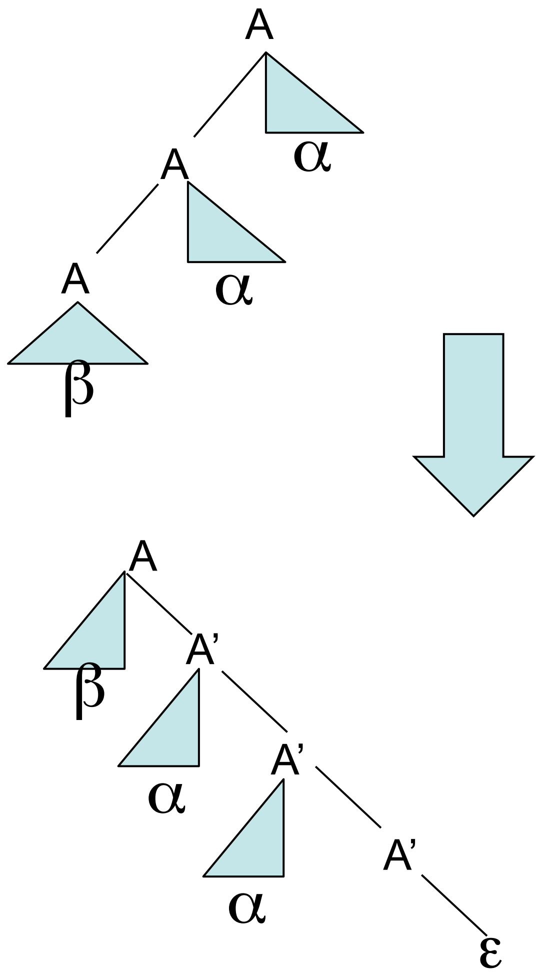
Eliminate Left-Recursion

Immediate left-recursion

A ::= Aα1 | Aα2 | . . . | Aαk |
        β1 | β2 | . . . | βn

A ::= β1 A’ | β2 A’ | . . . | βn A’

A’ ::= α1 A’ | α2 A’| . . . | αk A’ | ε



For eliminating left-recursion in general, see Aho and Ullman.

Eliminating Left Recursion

(G2)
S :: = E$

E ::= E + T
| E – T
| T

T ::= T * F
| T / F
| F

F ::= NUM
| ID
| (E)

Note that
E ::= T and
E ::= E + T
will cause problems
since FIRST(T) will be included
in FIRST(E + T) ---- so how can
we decide which poduction
To use based on next token?

Solution: eliminate “left recursion”!

E ::= T E’

E’ ::= + T E’
|

Eliminate left recursion

(G6)
S :: = E$

E ::= T E’

E’ ::= + T E’
| – T E’
|

T ::= F T’

T’ ::= * F T’
| / F T’
|

F ::= NUM
| ID
| (E)

FIRST and FOLLOW

For each non-terminal X we need to compute

FIRST[X] = the set of terminal symbols that can begin strings derived from X

FOLLOW[X] = the set of terminal symbols that can immediately follow X in some derivation

nullable[X] = true of X can derive the empty string, false otherwise

nullable[Z] = false, for Z in T

nullable[Y1 Y2 … Yk] = nullable[Y1] and … nullable[Yk], for Y(i) in N union T.

FIRST[Z] = {Z}, for Z in T

FIRST[X Y1 Y2 … Yk] = FIRST[X] if not nullable[X]

FIRST[X Y1 Y2 … Yk] =FIRST[X] union FIRST[Y1 … Yk] otherwise

Computing First, Follow, and nullable

```
For each terminal symbol Z
   FIRST[Z] := {Z};
   nullable[Z] := false;

For each non-terminal symbol X
  FIRST[X] := FOLLOW[X] := {};
  nullable[X] := false;

repeat
   for each production X → Y1 Y2 … Yk
      if Y1, … Yk are all nullable, or k = 0
         then nullable[X] := true
      for each i from 1 to k, each j from i + l to k
         if Y1 … Y(i-1) are all nullable or i = 1
            then  FIRST[X] := FIRST[X] union FIRST[Y(i)]
         if Y(i+1) … Yk are all nullable or if i = k
            then FOLLOW[Y(i)] := FOLLOW[Y(i)] union FOLLOW[X]
         if Y(i+1) … Y(j-1) are all nullable or i+1 = j
            then FOLLOW[Y(i)] := FOLLOW[Y(i)] union FIRST[Y(j)]
until there is no change
```

First, Follow, nullable table for G6

	Nullable	FIRST	FOLLOW
S	False	{ (, ID, NUM }	{}
E	False	{ (, ID, NUM }	{), $ }
E’	True	{ +, - }	{), $ }
T	False	{ (, ID, NUM }	{), +, -, $ }
T’	True	{ *, / }	{), +, -, $ }
F	False	{ (, ID, NUM }	{), *, /, +, -, $ }

(G6)
S :: = E$

E ::= T E’

E’ ::= + T E’
| – T E’
|

T ::= F T’

T’ ::= * F T’
| / F T’
|

F ::= NUM
| ID
| (E)

Predictive Parsing Table for G6

Table[X, T] = Set of productions

X ::= Y1…Yk in Table[X, T]
if T in FIRST[Y1 … Yk]
or if (T in FOLLOW[X] and nullable[Y1 … Yk])

NOTE: this could lead to more than one entry! If so, out of luck --- can't do recursive descent parsing!

	+	*	(	)	ID	NUM	$
S			S ::= E$		S ::= E$	S ::= E$	
E			E ::= T E'		E ::= T E'	E ::= T E'	
E'	E' ::= + T E'			E' ::=			E' ::=
T			T ::= F T'		T ::= F T'	T ::= F T'	
T'	T' ::=	T' ::= * F T'		T' ::=			T' ::=
F			F ::= (E)		F ::= ID	F ::= NUM	

(entries for /, - are similar…)

Left-most derivation is constructed by recursive descent

(G6)

S :: = E$

E ::= T E’

E’ ::= + T E’
| – T E’
|

T ::= F T’

T’ ::= * F T’
| / F T’
|

F ::= NUM
| ID
| (E)

Left-most derivation

S → E$
→ T E’ $
→ F T’ E’ $
→ (E) T’ E’ $
→ (T E’) T’ E’ $
→ (F T’ E’) T’ E’ $
→ (17 T’ E’) T’ E’ $
→ (17 E’) T’ E’ $
→ (17 + T E’) T’ E’ $
→ (17 + F T’ E’) T’ E’ $
→ (17 + 4 T’ E’) T’ E’ $
→ (17 + 4 E’) T’ E’ $
→ (17 + 4) T’ E’ $
→ (17 + 4) * F T’ E’ $
→ …
→ …
→ (17 + 4) * (2 – 10) T’ E’ $
→ (17 + 4) * (2 – 10) E’ $
→ (17 + 4) * (2 – 10)

call S()
on ‘(‘ call E()
on ‘(‘ call T()
.I..
…

As a stack machine

S → E$
→ T E’ $
→ F T’ E’ $
→ (E) T’ E’ $
→ (T E’) T’ E’ $
→ (F T’ E’) T’ E’ $
→ (17 T’ E’) T’ E’ $
→ (17 E’) T’ E’ $
→ (17 + T E’) T’ E’ $
→ (17 + F T’ E’) T’ E’ $
→ (17 + 4 T’ E’) T’ E’ $
→ (17 + 4 E’) T’ E’ $
→ (17 + 4) T’ E’ $
→ (17 + 4) * F T’ E’ $
→ …
→ …
→ (17 + 4) * (2 – 10) T’ E’ $
→ (17 + 4) * (2 – 10) E’ $
→ (17 + 4) * (2 – 10)

	E$
	T E’ $
	F T’ E’ $
(	E) T’ E’ $
(	T E’) T’ E’ $
(	F T’ E’) T’ E’ $
(17	T’ E’) T’ E’ $
(17	E’) T’ E’ $
(17 +	T E’) T’ E’ $
(17 +	F T’ E’) T’ E’ $
(17 + 4	T’ E’) T’ E’ $
(17 + 4	E’) T’ E’ $
(17 + 4)	T’ E’ $
(17 + 4) *	F T’ E’ $
…	
…	
(17 + 4) * (2 – 10)	T’ E’ $
(17 + 4) * (2 – 10)	E’ $
(17 + 4) * (2 – 10)	

But wait! What if there are conflicts in the predictive parsing table?

(G7)

S :: = d | X Y S

Y ::= c |

X ::= Y | a

	Nullable	FIRST	FOLLOW
S	false	{ c,d ,a}	{ }
Y	true	{ c }	{ c,d,a }
X	true	{ c,a }	{ c, a,d }

The resulting "predictive" table is not so predictive....

	a	c	d
S	{ S ::= X Y S }	{ S ::= X Y S }	{ S ::= X Y S, S ::= d }
Y	{ Y ::= }	{ Y ::= , Y ::= c}	{ Y ::= }
X	{ X ::= a, X ::= Y }	{ X ::= Y }	{ X ::= Y }

LL(1), LL(k), LR(0), LR(1), ...

- LL(k) : (L)eft-to-right parse, (L)eft-most derivation, k-symbol lookahead. Based on looking at the next k tokens, an LL(k) parser must *predict* the next production. We have been looking at LL(1).
- LR(k) : (L)eft-to-right parse, (R)ight-most derivation, k-symbol lookahead. Postpone production selection until *the entire* right-hand-side has been seen (and as many as k symbols beyond).
- LALR(1) : A special subclass of LR(1).

Example

(G8)

S :: = S ; S | ID = E | print (L)

E ::= ID | NUM | E + E | (S, E)

L ::= E | L, E

To be consistent, I should write the following, but I won’t…

(G8)

S :: = S SEMI S | ID EQUAL E | PRINT LPAREN L RPAREN

E ::= ID | NUM | E PLUS E | LPAREN S COMMA E RPAREN

L ::= E | L COMMA E

A right-most derivation ...

(G8)

S ::= S ; S
    | ID = E
    | print (L)

E ::= ID
    | NUM
    | E + E
    | (S, E)

L ::= E
    | L, E

<u>S</u>
→ S ; <u>S</u>
→ S ; ID = <u>E</u>
→ S ; ID = E + <u>E</u>
→ S ; ID = E + (S, <u>E</u>)
→ S ; ID = E + (S, <u>ID</u>)
→ S ; ID = E + (<u>S</u>, d)
→ S ; ID = E + (ID = <u>E</u>, d)
→ S ; ID = E + (ID = E + <u>E</u>, d)
→ S ; ID = E + (ID = E + <u>NUM</u>, d)
→ S ; ID = E + (ID = <u>E</u> + 6, d)
→ S ; ID = E + (ID = <u>NUM</u> + 6, d)
→ S ; ID = E + (<u>ID</u> = 5 + 6, d)
→ S ; ID = <u>E</u> + (d = 5 + 6, d)
→ S ; ID = <u>ID</u> + (d = 5 + 6, d)
→ S ; <u>ID</u> = c + (d = 5 + 6, d)
→ <u>S</u> ; b = c + (d = 5 + 6, d)
→ ID = <u>E</u> ; b = c + (d = 5 + 6, d)
→ ID = <u>NUM</u> ; b = c + (d = 5 + 6, d)
→ <u>ID</u> = 7 ; b = c + (d = 5 + 6, d)
→ a = 7 ; b = c + (d = 5 + 6, d)

Now, turn it upside down …

→ a = 7 ; b = c + (d = 5 + 6, d)
→ ID = 7 ; b = c + (d = 5 + 6, d)
→ ID = NUM; b = c + (d = 5 + 6, d)
→ ID = E ; b = c + (d = 5 + 6, d)
→ S ; b = c + (d = 5 + 6, d)
→ S ; ID = c + (d = 5 + 6, d)
→ S ; ID = ID + (d = 5 + 6, d)
→ S ; ID = E + (d = 5 + 6, d)
→ S ; ID = E + (ID = 5 + 6, d)
→ S ; ID = E + (ID = NUM + 6, d)
→ S ; ID = E + (ID = E + 6, d)
→ S ; ID = E + (ID = E + NUM, d)
→ S ; ID = E + (ID = E + E, d)
→ S ; ID = E + (ID = E, d)
→ S ; ID = E + (S, d)
→ S ; ID = E + (S, ID)
→ S ; ID = E + (S, E)
→ S ; ID = E + E
→ S ; ID = E
→ S ; S
S

Now, slice it down the middle...

	a = 7 ; b = c + (d = 5 + 6, d)
ID	= 7 ; b = c + (d = 5 + 6, d)
ID = NUM	; b = c + (d = 5 + 6, d)
ID = E	; b = c + (d = 5 + 6, d)
S	; b = c + (d = 5 + 6, d)
S ; ID	= c + (d = 5 + 6, d)
S ; ID = ID	+ (d = 5 + 6, d)
S ; ID = E	+ (d = 5 + 6, d)
S ; ID = E + (ID	= 5 + 6, d)
S ; ID = E + (ID = NUM	+ 6, d)
S ; ID = E + (ID = E	+ 6, d)
S ; ID = E + (ID = E + NUM	, d)
S ; ID = E + (ID = E + E	, d)
S ; ID = E + (ID = E	, d)
S ; ID = E + (S	, d)
S ; ID = E + (S, ID	)
S ; ID = E + (S, E)	
S ; ID = E + E	
S ; ID = E	
S ; S	
S	

The rest of the input string

A stack of terminals and non-terminals

Now, add some actions. s = SHIFT, r = REDUCE

	a = 7 ; b = c + (d = 5 + 6, d)	s
ID	= 7 ; b = c + (d = 5 + 6, d)	s, s
ID = NUM	; b = c + (d = 5 + 6, d)	r E ::= NUM
ID = E	; b = c + (d = 5 + 6, d)	r S ::= ID = E
S	; b = c + (d = 5 + 6, d)	s, s
S ; ID	= c + (d = 5 + 6, d)	s, s
S ; ID = ID	+ (d = 5 + 6, d)	r E ::= ID
S ; ID = E	+ (d = 5 + 6, d)	s, s, s
S ; ID = E + (ID	= 5 + 6, d)	s, s
S ; ID = E + (ID = NUM	+ 6, d)	r E ::= NUM
S ; ID = E + (ID = E	+ 6, d)	s, s
S ; ID = E + (ID = E + NUM	, d)	r E ::= NUM
S ; ID = E + (ID = E + E	, d)	r E ::= E+E, s, s
S ; ID = E + (ID = E	, d)	r S ::= ID = E
S ; ID = E + (S	)	R E::= ID
S ; ID = E + (S, ID	)	s, r E ::= (S, E)
S ; ID = E + (S, E)		r E ::= E + E
S ; ID = E + E		r S ::= ID = E
S ; ID = E		r S ::= S ; S
S ; S		
S		

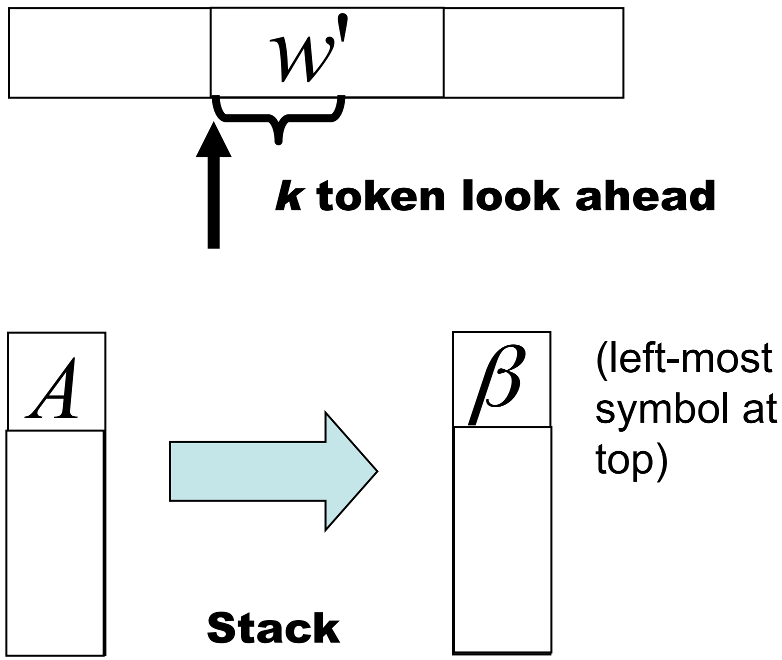
SHIFT = LEX + move token to stack

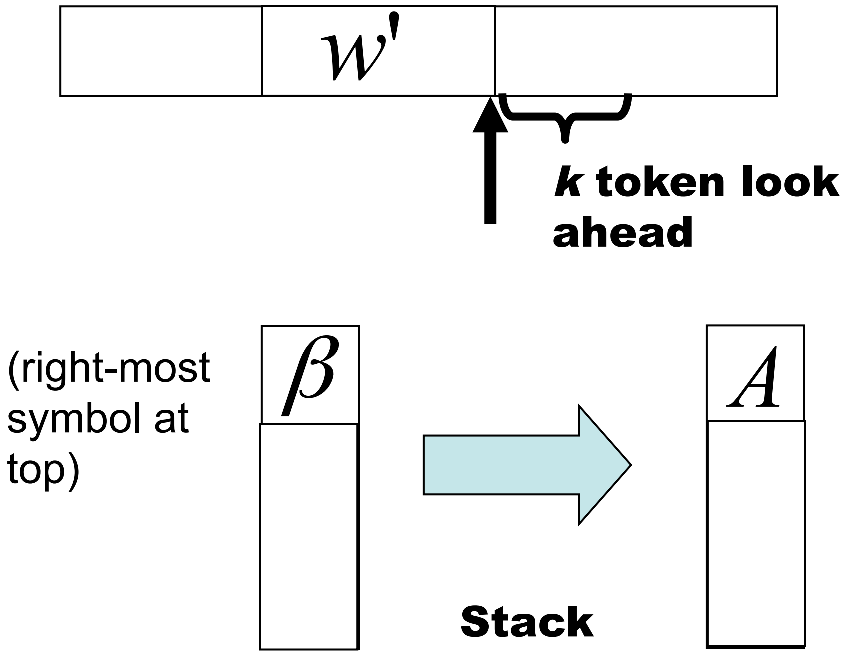
ACTIONS

LL(k) vs. LR(k) reductions

$$A \rightarrow \beta \Rightarrow^{*} w' \qquad (\beta \in (T \cup N)^{*}, \quad w' \in T^{*})$$

$LL(k)$	$LR(k)$





The language of this Stack IS REGULAR!

Q: How do we know when to shift and when to reduce? A: Build a FSA from LR(0) Items!

(G10)

S ::= A $

A ::= (A)
  | ( )

If

X ::= $\alpha\beta$

is a production, then

X ::= $\alpha \bullet \beta$

is an LR(0) item.

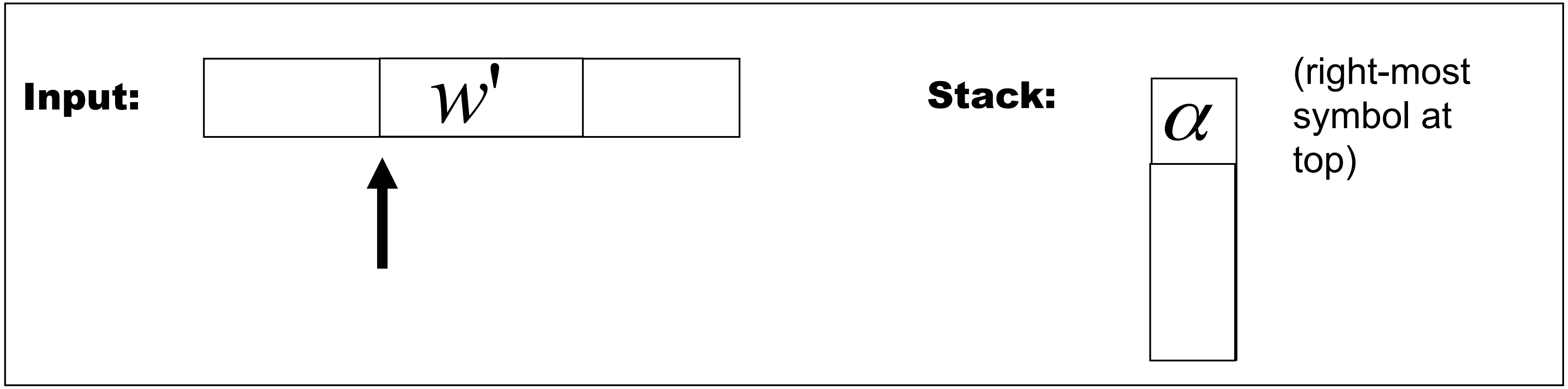
S ::= • A $
S ::= A • $
A ::= • (A)
A ::= (• A)
A ::= (A •)
A ::= (A) •
A ::= • ( )
A ::= (•)
A ::= ( ) •

LR(0) items indicate what is on the stack (to the left of the •) and what is still in the input stream (to the right of the •)

LR(k) states (non-deterministic)

The state $$(A \rightarrow \alpha \bullet \beta,\ a_1 a_2 \cdots a_k)$$

should represent this situation:



with $$\beta a_1 a_2 \cdots a_k \Rightarrow^* w'$$

Key idea behind LR(0) items

- If the "current state" contains the item $A ::= \alpha \bullet c\ \beta$ and the current symbol in the input buffer is c
  - the state prompts parser to perform a shift action
  - next state will contain $A ::= \alpha\ c \bullet \beta$
- If the "state" contains the item $\mathbf{A ::= \alpha \bullet}$
  - the state prompts parser to perform a reduce action
- If the "state" contains the item $S ::= \alpha \bullet \$$ and the input buffer is empty
  - the state prompts parser to accept
- But How about $A ::= \alpha \bullet X\ \beta$ where X is a nonterminal?

The NFA for LR(0) items

- The transition of LR(0) items can be represented by an NFA, in which
  - 1. each LR(0) item is a state,
  - 2. there is a transition from item $A ::= \alpha \bullet c\ \beta$

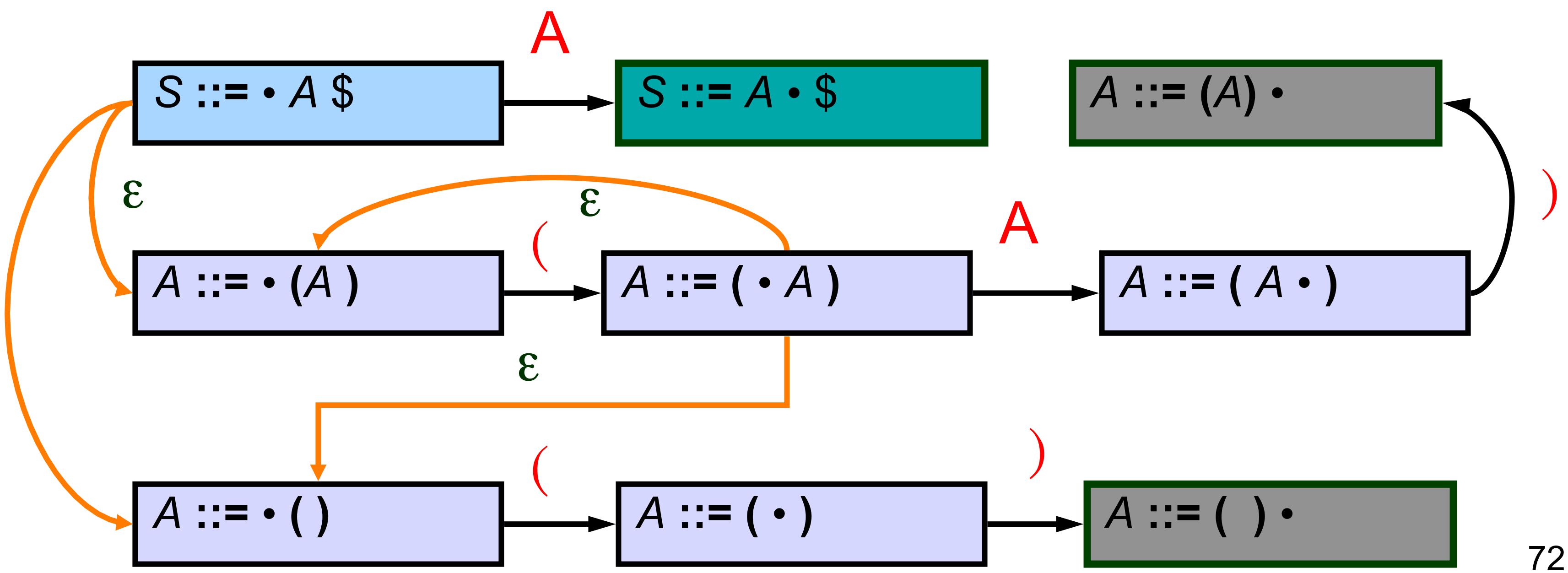
    to item $A ::= \alpha c \bullet \beta$ with label c, where c is a terminal symbol
  - 3. there is an ε-transition from item $A ::= \alpha \bullet X\ \beta$ to $X ::= \bullet\ \gamma$, where X is a non-terminal
  - 4. $S ::= \bullet\ A\ \$$ is the start state
  - 5. $A ::= \alpha \bullet$ is a final state.

Example NFA for Items

S ::= • A $ S ::= A • $ A ::= • (A)

A ::= (• A)   A ::= (A •)   A ::= (A) •

A ::= • ( )   A ::= (•)   A ::= ( ) •



The DFA from LR(0) items

- After the NFA for LR(0) is constructed, the resulting DFA for LR(0) parsing can be obtained by the usual NFA2DFA construction.

- we thus require
  - ε-closure (I)
  - move(S, a)

Fixed Point Algorithm for Closure**(I)**

- Every item in **I** is also an item in Closure**(I)**
- If A ::= $\alpha \bullet B \beta$ is in Closure**(I)** and B ::= $\bullet \gamma$ is an item, then add B ::= $\bullet \gamma$ to Closure**(I)**
- Repeat until no more new items can be added to Closure**(I)**

Examples of Closure

$\text{Closure}(\{A ::= (\bullet A)\}) =$

$$\left\{\begin{array}{l} A ::= (\bullet A) \\ A ::= \bullet (A) \\ A ::= \bullet (\ ) \end{array}\right\}$$

- $\text{closure}(\{S ::= \bullet \boldsymbol{A}\ \$\})$

$$\left\{\begin{array}{l} S ::= \bullet \boldsymbol{A}\ \$ \\ \boldsymbol{A} ::= \bullet (\boldsymbol{A}) \\ \boldsymbol{A} ::= \bullet (\ ) \end{array}\right\}$$

S ::= • A $
S ::= A • $
A ::= • (A)
A ::= (• A)
A ::= (A •)
A ::= (A) •
A ::= • ( )
A ::= (•)
A ::= ( ) •

Goto() of a set of items

- Goto finds the new state after consuming a grammar symbol while in the current state

- Algorithm for *Goto*(I, X)
  where I is a set of items
  and X is a non-terminal

$$\text{Goto(I, X)} = \text{Closure}\left(\left\{ \text{A} ::= \alpha\ \text{X} \bullet \beta \;\middle|\; \text{A} ::= \alpha \bullet \text{X}\ \beta \text{ in I} \right\} \right)$$

- goto is the new set obtained by "moving the dot" over X

Examples of Goto

- Goto ({*A* ::= •(*A*)}, ()

$$\left\{\begin{array}{l} A ::= (\ \bullet A) \\ A ::= \bullet (A) \\ A ::= \bullet (\ ) \end{array}\right\}$$

- Goto ({*A* ::= (• *A*)}, *A*)

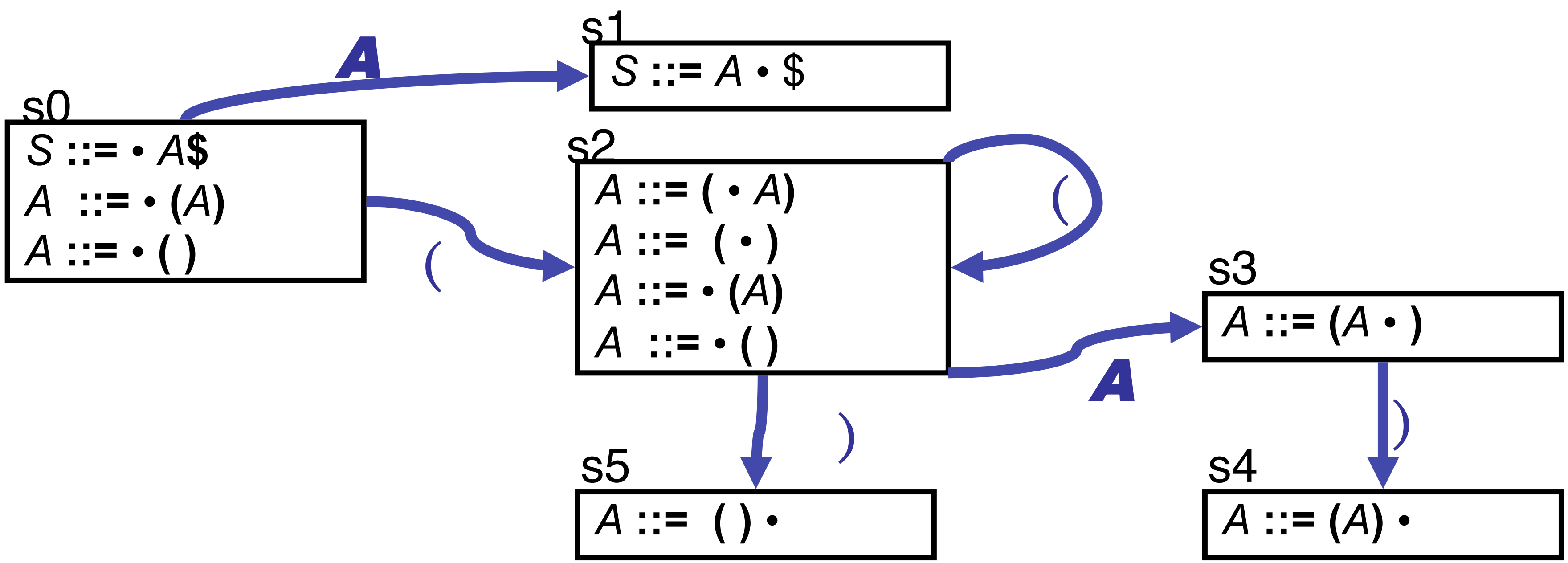
$$\left\{ A ::= (A \bullet\ ) \right\}$$

S ::= • A $
S ::= A • $
A ::= • (A)
A ::= (• A)
A ::= (A •)
A ::= (A) •
A ::= • ( )
A ::= (•)
A ::= ( ) •

Building the DFA states

- Essentially the usual NFA2DFA construction!!
- Let A be the start symbol and S a new start symbol.
- Create a new rule S ::= A $
- Create the first state to be Closure({ *S* **::=** • ***A*** $})
- Pick a state **I**
  - for each item A **::=** α • X β in **I**
    - find Goto**(I, X)**
    - if Goto**(I, X)** is not already a state, make one
    - Add an edge X from state **I** to Goto**(I, X)** state
- Repeat until no more additions possible

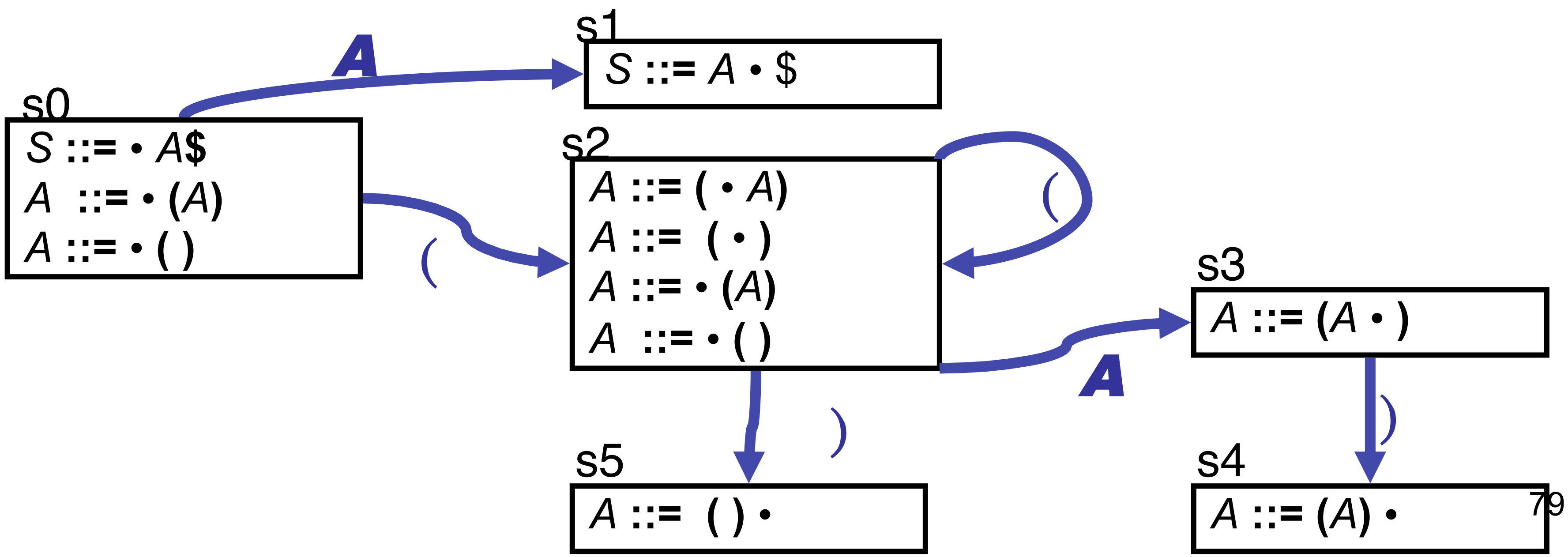
DFA Example



Creating the Parse Table(s)

(G10)
(1) S ::= A$
(2) A ::= (A)
(3) A ::= ( )

State	(	)	$	A
s0	shift to s2			goto s1
s1			accept	
s2	shift to s2	shift to s5		goto s3
s3		shift to s4		
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	



Parsing with an LR Table

Use table and top-of-stack and input symbol to get action:

If action is

shift sn : advance input one token,
push sn on stack

reduce X ::= α : pop stack 2* $|\alpha|$ times (grammar symbols are paired with states). In the state now on top of stack, use goto table to get next state sn, push it on top of stack

accept : stop and accept

error : weep (actually, produce a good error message)

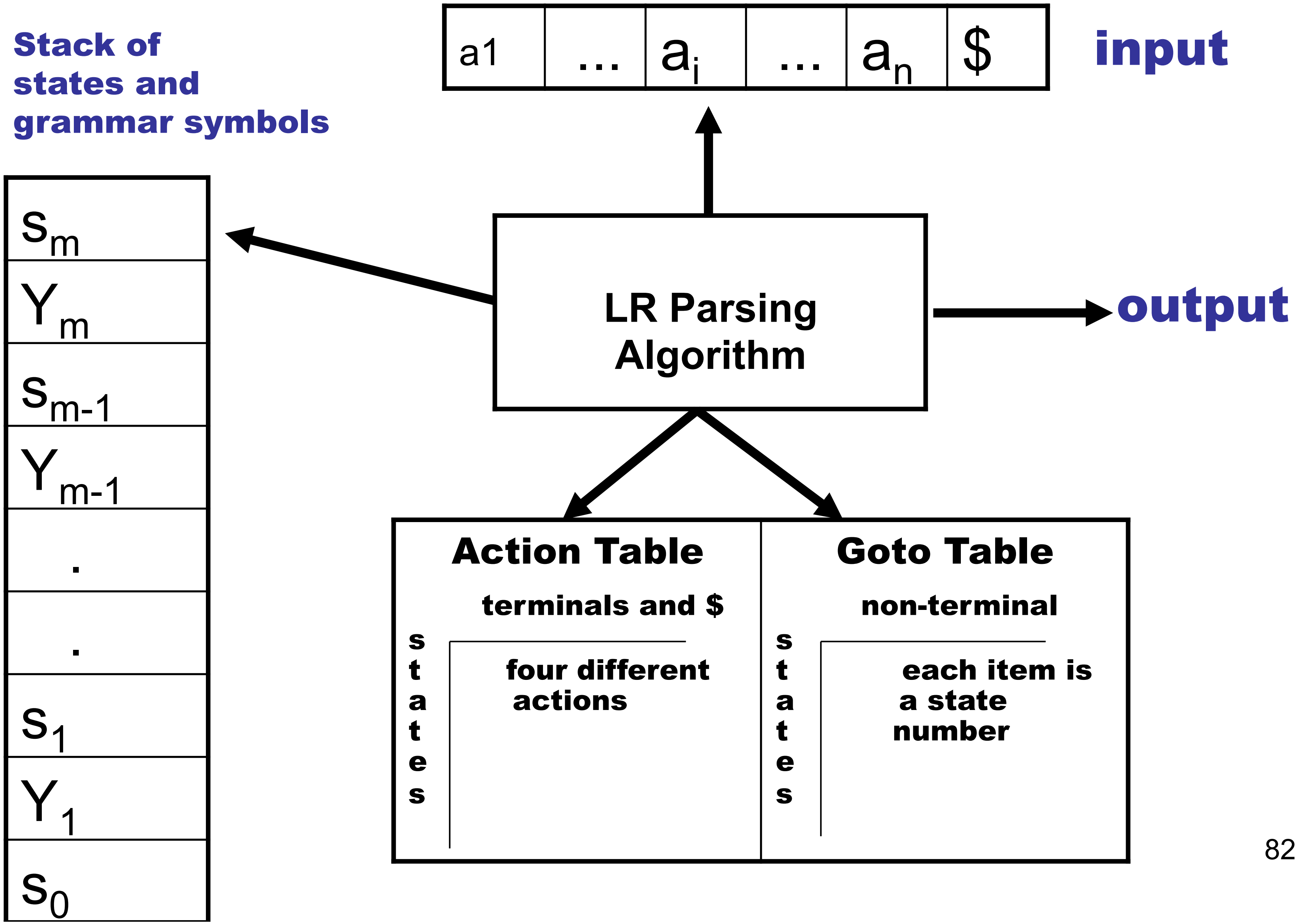
Parsing, again…

(G10)
(1) S ::= A$
(2) A ::= (A)
(3) A ::= ( )

		ACTION		Goto
State	(	)	$	A
s0	shift to s2			goto s1
s1			accept	
s2	shift to s2	shift to s5		goto s3
s3		shift to s4		
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

s0	**(())$**	**shift s2**
s0 (s2	**())$**	**shift s2**
s0 (s2 (s2	**))$**	**shift s5**
s0 (s2 (s2) s5	**)$**	**reduce A ::= ()**
s0 (s2 A	**)$**	**goto s3**
s0 (s2 A s3	**)$**	**shift s4**
s0 (s2 A s3) s4	**$**	**reduce A::= (A)**
s0 A	**$**	**goto s1**
s0 A s1	**$**	**ACCEPT!**

LR Parsing Algorithm



Problem With LR(0) Parsing

- **No lookahead**
- **Vulnerable to unnecessary conflicts**
  - **Shift/Reduce Conflicts (may reduce too soon in some cases)**
  - **Reduce/Reduce Conflicts**
- **Solutions:**
  - **LR(1) parsing - systematic lookahead**

LR(1) Items

- An LR(1) item is a pair:

  $(X ::= \alpha\, .\, \beta,\ a)$

  – $X ::= \alpha\beta$ is a production
  – a is a terminal (the lookahead terminal)
  – LR(1) means 1 lookahead terminal

- $[X ::= \alpha\, .\, \beta,\ a]$ describes a context of the parser
  – We are trying to find an X followed by an a, and
  – We have (at least) α already on top of the stack
  – Thus we need to see next a prefix derived from βa

The Closure Operation

- Need to modify closure operation:.

Closure(Items) =
  repeat
    for each [X ::= α . Yβ, a] in Items
      for each production Y ::= γ
        for each b in First(βa)
          add [Y ::= .γ, b] to Items
  until Items is unchanged

Constructing the Parsing DFA (2)

- A DFA state is a closed set of LR(1) items
- The start state contains (S' ::= .S$, dummy)
- A state that contains [X ::= α., b] is labeled with "reduce with X ::= α on lookahead b"
- And now the transitions …

The DFA Transitions

- A state s that contains [X ::= α.Yβ, b] has a transition labeled y to the state obtained from Transition(s, Y)
  - Y can be a terminal or a non-terminal

Transition(s, Y)

  Items = {}

  for each [X ::= α.Yβ, b] in s

      add [X ! αY.β, b] to Items

  return Closure(Items)

LR(1)-the parse table

- Shift and goto as before
- Reduce
  - state I with item $(A \rightarrow \alpha., z)$ gives a reduce $A \rightarrow \alpha$ if z is the next character in the input.

- LR(1)-parse tables are very big

LR(1)-DFA

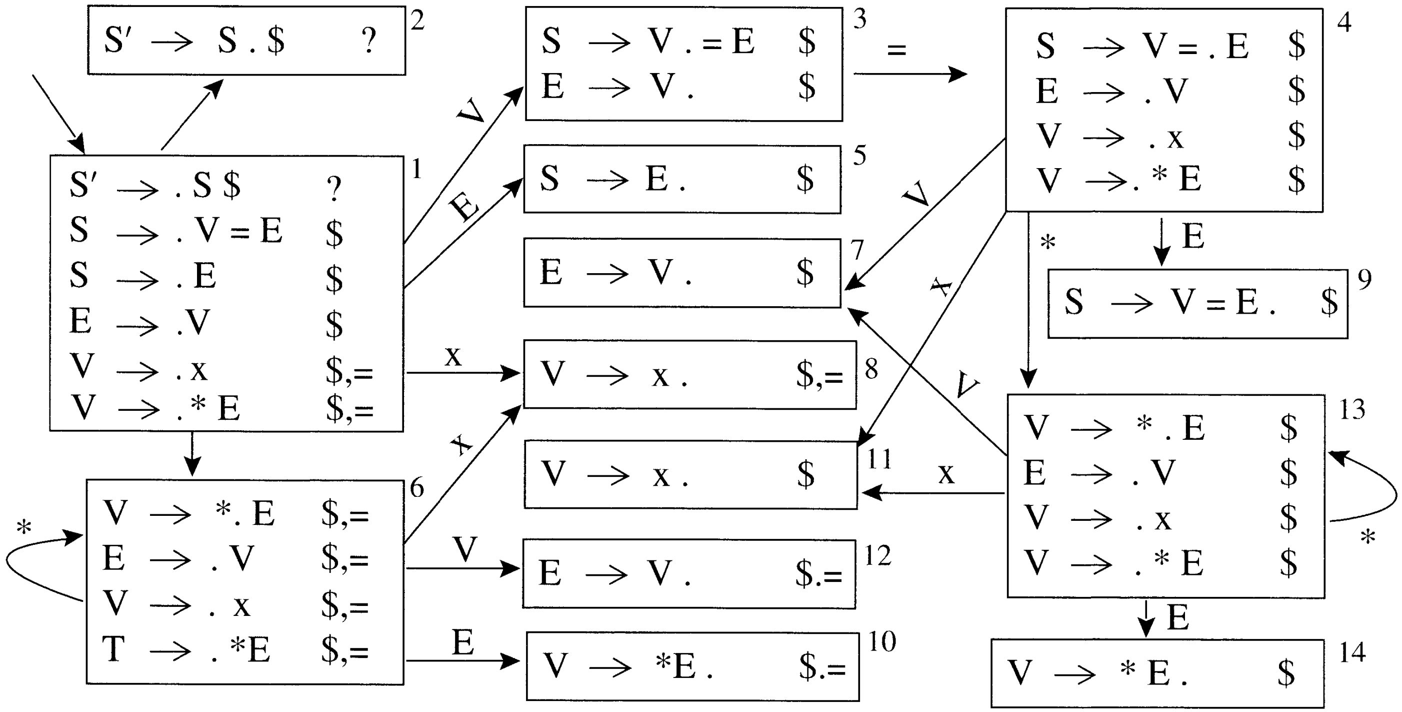
(G11)

S' ::= S$

S ::= V = E
| E

E ::= V

V ::= x
| *E



From Andrew Appel, “Modern Compiler Implementation in Java” page 65

LR(1)-parse table

	x	*	=	$	S	E	V		x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3	8			r4	r4			
2				acc				9				r1			
3			s4	r3				10			r5	r5			
4	s11	s13				g9	g7	11				r4			
5				r2				12			r3	r3			
6	s8	s6				g10	g12	13	s11	s13				g14	g7
7				r3				14				r5			

LALR States

- Consider for example the LR(1) states

$$\{[X ::= \alpha. , a], [Y ::= \beta. , c]\}$$

$$\{[X ::= \alpha. , b], [Y ::= \beta. , d]\}$$

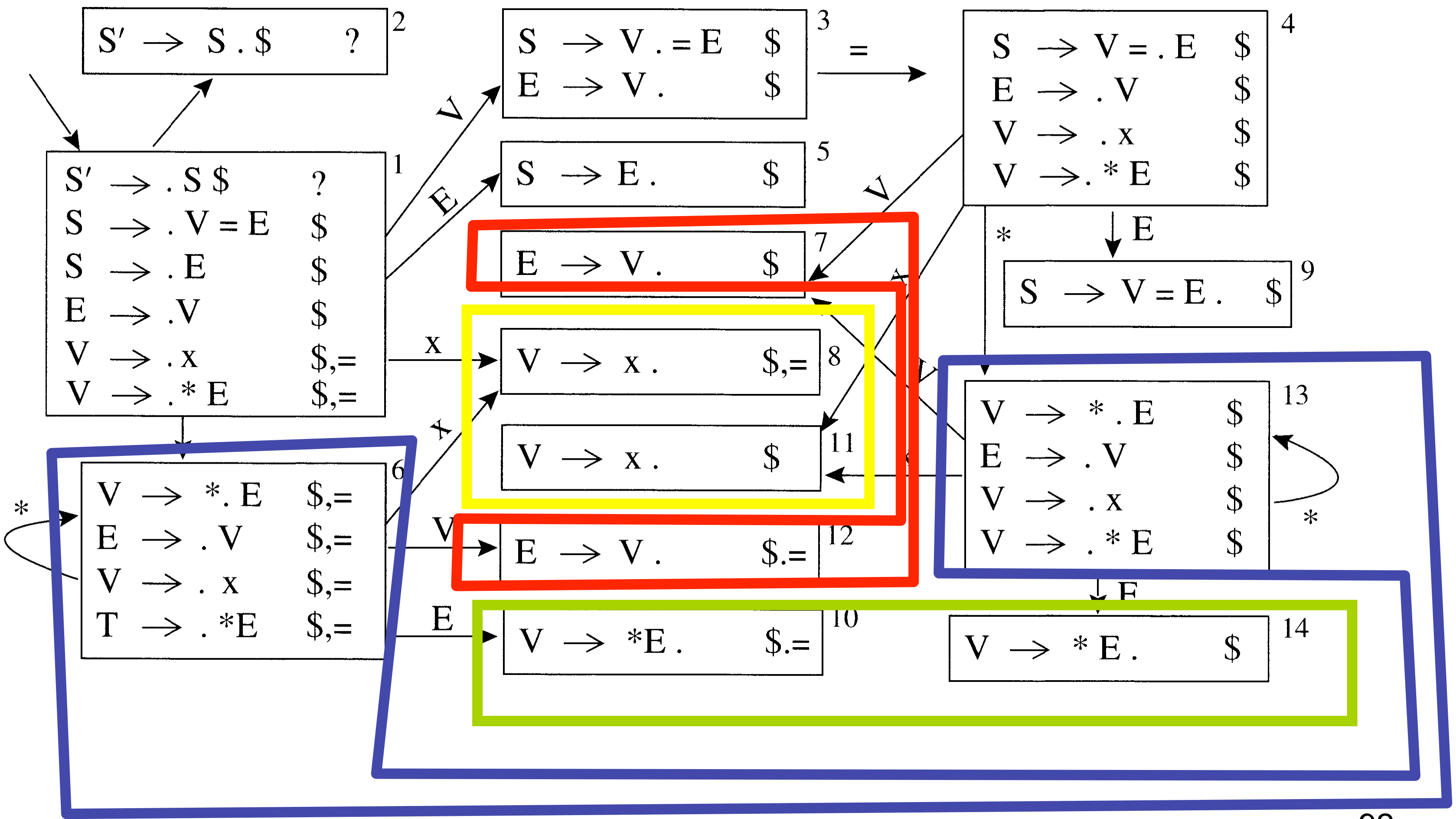
- They have the same <u>*core*</u> and can be merged to the state

$$\{[X ::= \alpha. , a/b], [Y ::= \beta. , c/d]\}$$

- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)

For LALR(1), Collapse States ...

Combine states 6 and 13, 7 and 12, 8 and 11, 10 and 14.



LALR(1)-parse-table

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				acc			
3			s4	r3			
4	s8	s6				g9	g7
5							
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

LALR vs. LR Parsing

- LALR languages are not "natural"
  - They are an efficiency hack on LR languages

- You may see claims that any reasonable programming language has a LALR(1) grammar, {Arguably this is done by defining languages without an LALR(1) grammar as unreasonable ☺ }.

- In any case, LALR(1) has become a standard for programming languages and for parser generators, in spite of its apparent complexity.