Review for the Midterm

COMS W4115

Prof. Stephen A. Edwards
Fall 2004
Columbia University
Department of Computer Science

The Midterm

70 minutes

4–5 problems

Closed book

One sheet of notes of your own devising

Comprehensive: Anything discussed in class is fair game

Little, if any, programming.

Details of ANTLR/C/Java/Prolog/ML syntax not required

Broad knowledge of languages discussed

Topics

Structure of a Compiler

Scripting Languages

Scanning and Parsing

Regular Expressions

Context-Free Grammars

Top-down Parsing

Bottom-up Parsing

ASTs

Name, Scope, and Bindings

Control-flow constructs

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

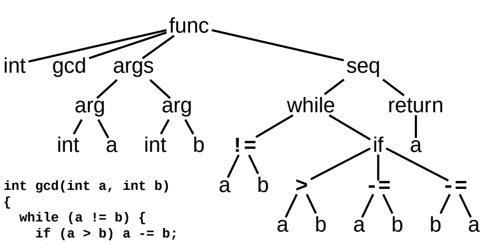
Lexical Analysis Gives Tokens


```
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.

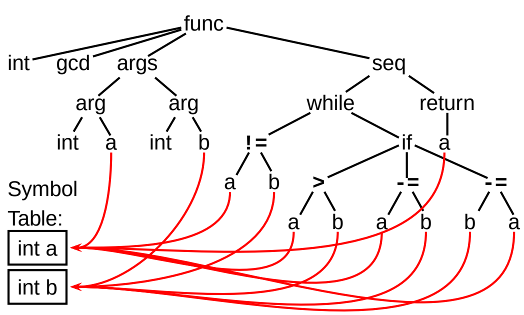
Parsing Gives an AST




```
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.

Semantic Analysis Resolves Symbols



Types checked; references to symbols resolved

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

Generation of 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
```

Scanning and Automata

Describing Tokens

Alphabet: A finite set of symbols

Examples: { 0, 1 }, { A, B, C, ..., Z }, ASCII, Unicode

String: A finite sequence of symbols from an alphabet

Examples: ϵ (the empty string), Stephen, $\alpha\beta\gamma$

Language: A set of strings over an alphabet

Examples: $\emptyset$ (the empty language), { 1, 11, 111, 1111 }, all English words, strings that start with a letter followed by any sequence of letters and digits

Operations on Languages

Let $L = \{ \epsilon, \text{wo} \}$, $M = \{ \text{man, men} \}$

Concatenation: Strings from one followed by the other

$LM = \{ \text{man, men, woman, women} \}$

Union: All strings from each language

$L \cup M = \{\epsilon, \text{wo, man, men} \}$

Kleene Closure: Zero or more concatenations

$M^* = \{\epsilon, M, MM, MMM, \ldots\} =$
$\{\epsilon$, man, men, manman, manmen, menman, menmen, manmanman, manmanmen, manmenman, $\ldots\}$

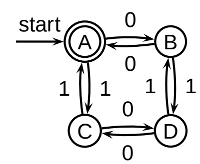
Regular Expressions over an Alphabet Σ

A standard way to express languages for tokens.

1. ϵ is a regular expression that denotes $\{\epsilon\}$
2. If $a \in \Sigma$, a is an RE that denotes $\{a\}$
3. If r and s denote languages $L(r)$ and $L(s)$,
   - $(r)|(s)$ denotes $L(r) \cup L(s)$
   - $(r)(s)$ denotes $\{tu : t \in L(r), u \in L(s)\}$
   - $(r)^*$ denotes $\cup_{i=0}^{\infty} L^i$ ($L^0 = \emptyset$ and $L^i = LL^{i-1}$)

Nondeterministic Finite Automata

"All strings containing an even number of 0's and 1's"



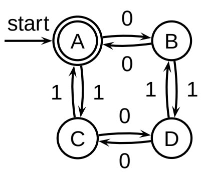
1. Set of states S: {(A), B, C, D}
2. Set of input symbols Σ: $\{0, 1\}$
3. Transition function $\sigma : S \times \Sigma_\epsilon \to 2^S$

state	ϵ	0	1
A	–	{B}	{C}
B	–	{A}	{D}
C	–	{D}	{A}
D	–	{C}	{B}

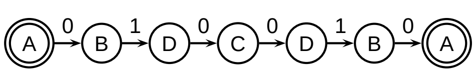
4. Start state s_0 : A
5. Set of accepting states F: {(A)}

The Language induced by an NFA

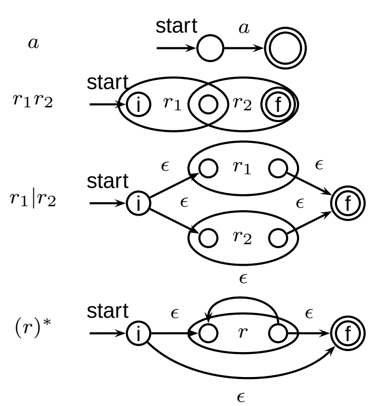
An NFA accepts an input string x iff there is a path from the start state to an accepting state that "spells out" x.



Show that the string "010010" is accepted.

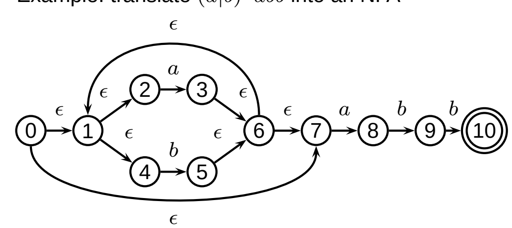


Translating REs into NFAs

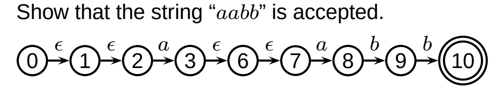


Translating REs into NFAs

Example: translate $(a|b)^*abb$ into an NFA



Show that the string "$aabb$" is accepted.



Simulating NFAs

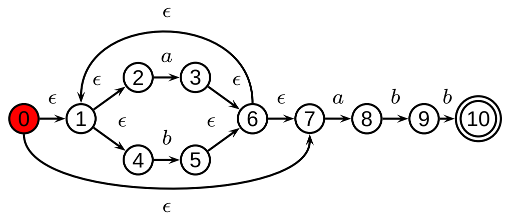
Problem: you must follow the “right” arcs to show that a string is accepted. How do you know which arc is right?

Solution: follow them all and sort it out later.

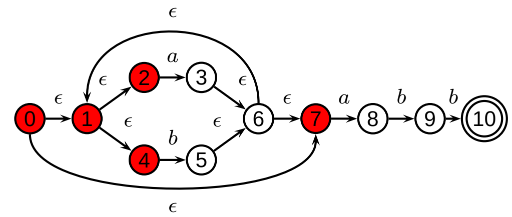
“Two-stack” NFA simulation algorithm:

1. Initial states: the ϵ-closure of the start state
2. For each character c,
   - New states: follow all transitions labeled c
   - Form the ϵ-closure of the current states
3. Accept if any final state is accepting

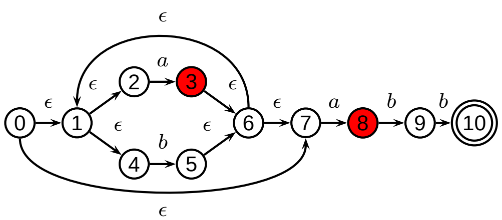
Simulating an NFA: $\cdot aabb$, Start



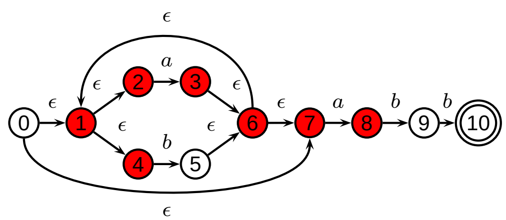
Simulating an NFA: $\cdot aabb$, ϵ-closure



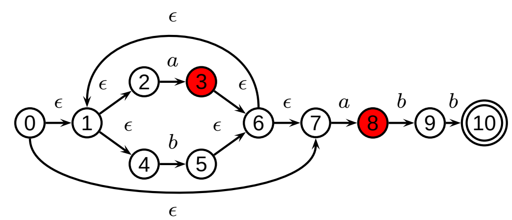
Simulating an NFA: $a\cdot abb$



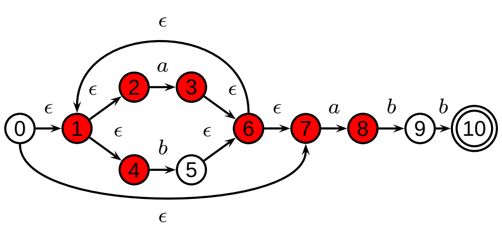
Simulating an NFA: $a\cdot abb$, ϵ-closure



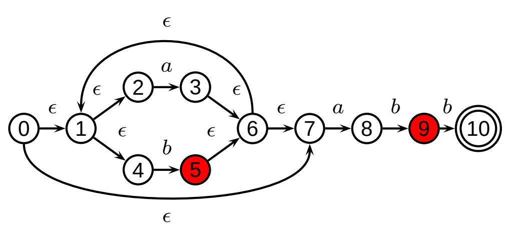
Simulating an NFA: $aa\cdot bb$



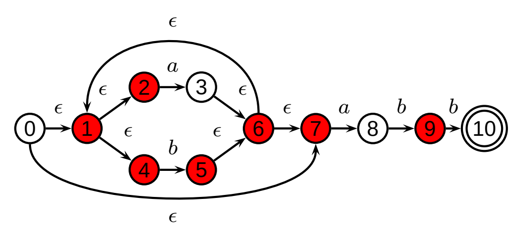
Simulating an NFA: $aa\cdot bb$, ϵ-closure



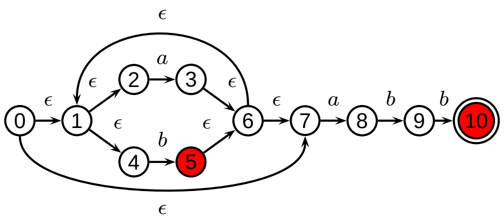
Simulating an NFA: $aab\cdot b$



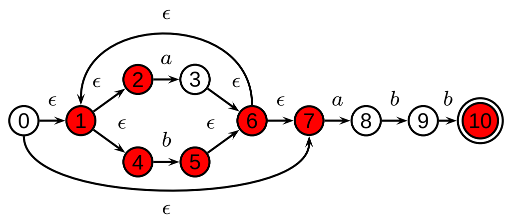
Simulating an NFA: $aab\cdot b$, ϵ-closure



Simulating an NFA: $aabb\cdot$



Simulating an NFA: $aabb\cdot$, Done



Deterministic Finite Automata

Restricted form of NFAs:

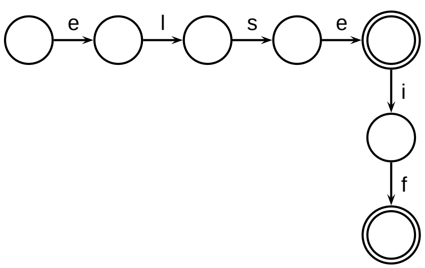
- No state has a transition on ϵ
- For each state s and symbol a, there is at most one edge labeled a leaving s.

Differs subtly from the definition used in COMS W3261 (Sipser, *Introduction to the Theory of Computation*)

Very easy to check acceptance: simulate by maintaining current state. Accept if you end up on an accepting state. Reject if you end on a non-accepting state or if there is no transition from the current state for the next symbol.

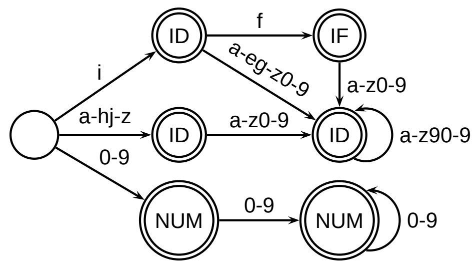
Deterministic Finite Automata

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



Deterministic Finite Automata

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



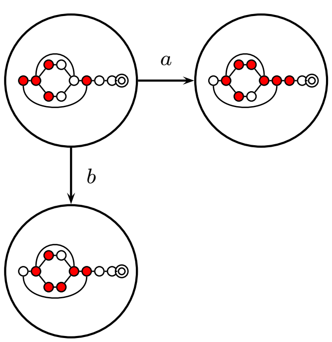
Building a DFA from an NFA

Subset construction algorithm

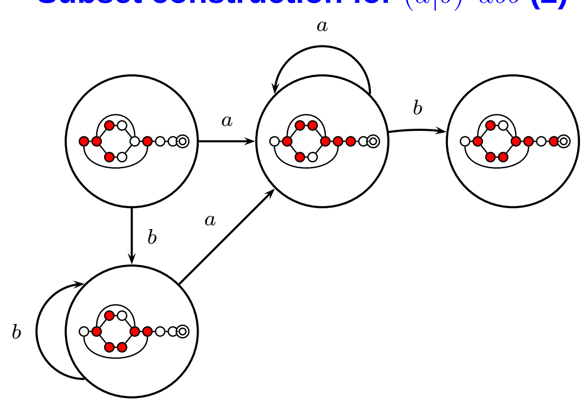
Simulate the NFA for all possible inputs and track the states that appear.

Each unique state during simulation becomes a state in the DFA.

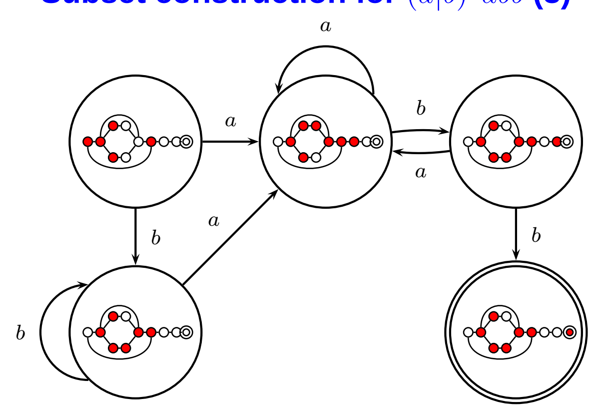
Subset construction for $(a|b)^*abb$ (1)



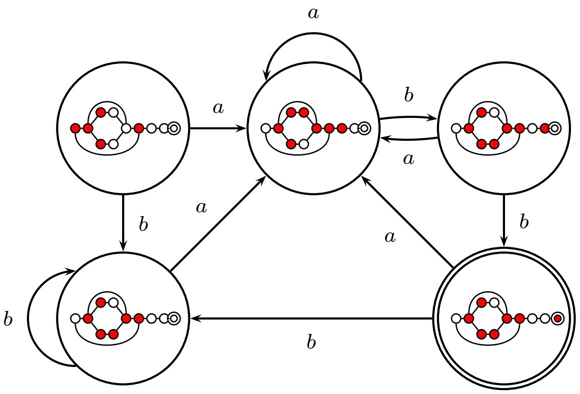
Subset construction for $(a|b)^*abb$ (2)



Subset construction for $(a|b)^*abb$ (3)



Subset construction for $(a|b)^*abb$ (4)



Grammars and Parsing

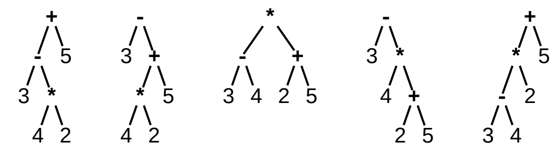
Ambiguous Grammars

A grammar can easily be ambiguous. Consider parsing

3 - 4 * 2 + 5

with the grammar

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



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 ;
```

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 */
      ;
```

Bottom-up Parsing

Rightmost Derivation

$1: \; e \to t + e$
$2: \; e \to t$
$3: \; t \to \textbf{Id} * t$
$4: \; t \to \textbf{Id}$

A rightmost derivation for **Id** $*$ **Id** $+$ **Id**:

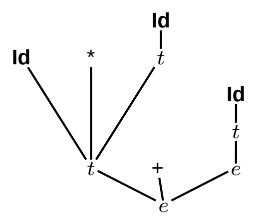
e
$t + e$
$t + t$
$t +$ **Id**
Id $* \, t +$ **Id**
Id $*$ **Id** $+$ **Id**

Basic idea of bottom-up parsing: construct this rightmost derivation backward.

Handles

$1: \; e \to t + e$
$2: \; e \to t$
$3: \; t \to \textbf{Id} * t$
$4: \; t \to \textbf{Id}$

Id $*$ **Id** $+$ **Id**
Id $* \, t +$ **Id**
$t +$ **Id**
$t + t$
$t + e$
e



This is a reverse rightmost derivation for **Id** $*$ **Id** $+$ **Id**.

Each highlighted section is a handle.

Taken in order, the handles build the tree from the leaves to the root.

Shift-reduce Parsing

$1: \; e \to t + e$
$2: \; e \to t$
$3: \; t \to \textbf{Id} * t$
$4: \; t \to \textbf{Id}$

stack	input	action
	Id $*$ **Id** $+$ **Id**	shift
Id	$*$ **Id** $+$ **Id**	shift
Id $*$	**Id** $+$ **Id**	shift
Id $*$ **Id**	$+$ **Id**	reduce (4)
Id $* \, t$	$+$ **Id**	reduce (3)
t	$+$ **Id**	shift
$t+$	**Id**	shift
$t +$ **Id**		reduce (4)
$t + t$		reduce (2)
$t + e$		reduce (1)
e		accept

Scan input left-to-right, looking for handles.

An oracle tells what to do

LR Parsing

$1: \; e \to t + e$
$2: \; e \to t$
$3: \; t \to \textbf{Id} * t$
$4: \; t \to \textbf{Id}$

	action				goto	
	Id	+	*	$	e	t
0	s1				7	2
1	r4	r4	s3	r4		
2	r2	s4	r2	r2		
3	s1					5
4	s1				6	2
5	r3	r3	r3	r3		
6	r1	r1	r1	r1		
7				acc		

stack	input	action
[0]	**Id** * **Id** + **Id** $	shift, goto 1

1. Look at state on top of stack
2. and the next input token
3. to find the next action
4. In this case, shift the token onto the stack and go to state 1.

LR Parsing

$1: \; e \to t + e$
$2: \; e \to t$
$3: \; t \to \textbf{Id} * t$
$4: \; t \to \textbf{Id}$

	action				goto	
	Id	+	*	$	e	t
0	s1				7	2
1	r4	r4	s3	r4		
2	r2	s4	r2	r2		
3	s1					5
4	s1				6	2
5	r3	r3	r3	r3		
6	r1	r1	r1	r1		
7				acc		

stack	input	action
[0]	**Id** * **Id** + **Id** $	shift, goto 1
[0] [Id 1]	* **Id** + **Id** $	shift, goto 3
[0] [Id 1] [* 3]	**Id** + **Id** $	shift, goto 1
[0] [Id 1] [* 3] [Id 1]	+ **Id** $	reduce w/ 4

Action is reduce with rule 4 ($t \to$ **Id**). The right side is removed from the stack to reveal state 3. The goto table in state 3 tells us to go to state 5 when we reduce a t:

stack	input	action
[0] [Id 1] [* 3] [t 5]	+ **Id** $	

LR Parsing

$1: \; e \to t + e$
$2: \; e \to t$
$3: \; t \to \textbf{Id} * t$
$4: \; t \to \textbf{Id}$

	action				goto	
	Id	+	*	$	e	t
0	s1				7	2
1	r4	r4	s3	r4		
2	r2	s4	r2	r2		
3	s1					5
4	s1				6	2
5	r3	r3	r3	r3		
6	r1	r1	r1	r1		
7				acc		

stack	input	action
[0]	**Id** * **Id** + **Id** $	shift, goto 1
[0] [Id 1]	* **Id** + **Id** $	shift, goto 3
[0] [Id 1] [* 3]	**Id** + **Id** $	shift, goto 1
[0] [Id 1] [* 3] [Id 1]	+ **Id** $	reduce w/ 4
[0] [Id 1] [* 3] [t 5]	+ **Id** $	reduce w/ 3
[0] [t 2]	+ **Id** $	shift, goto 4
[0] [t 2] [+ 4]	**Id** $	shift, goto 1
[0] [t 2] [+ 4] [Id 1]	$	reduce w/ 4
[0] [t 2] [+ 4] [t 2]	$	reduce w/ 2
[0] [t 2] [+ 4] [e 6]	$	reduce w/ 1
[0] [e 7]	$	accept

Constructing the SLR Parse Table

The states are places we could be in a reverse-rightmost derivation. Let's represent such a place with a dot.

$1: \quad e \to t + e$

$2: \quad e \to t$

$3: \quad t \to \textbf{Id} * t$

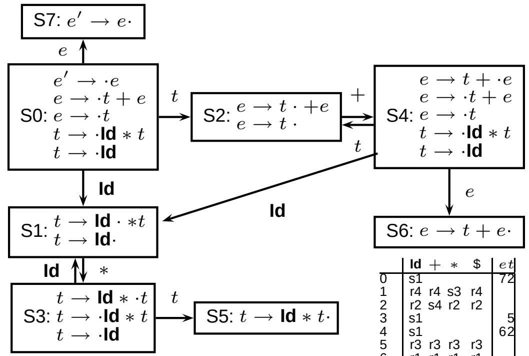
$4: \quad t \to \textbf{Id}$

Say we were at the beginning ($\cdot e$). This corresponds to

$$
\begin{aligned}
& e' \to \cdot e \\
& e \to \cdot t + e \\
& e \to \cdot t \\
& t \to \cdot \textbf{Id} * t \\
& t \to \cdot \textbf{Id}
\end{aligned}
$$

The first is a placeholder. The second are the two possibilities when we're just before e. The last two are the two possibilities when we're just before t.

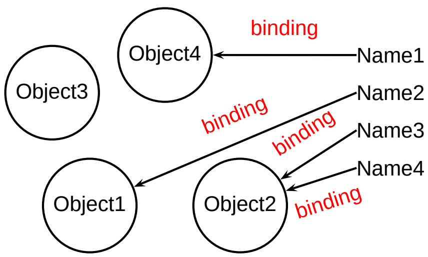
Constructing the SLR Parsing Table



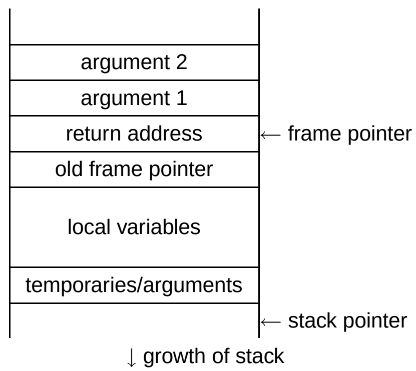
	Id	+	*	$	e	t
0	s1				7	2
1	r4	r4	s3	r4		
2	r2	s4	r2	r2		
3	s1					5
4	s1				6	2
5	r3	r3	r3	r3		
6	r1	r1	r1	r1		
7				acc		

Names, Objects, and Bindings

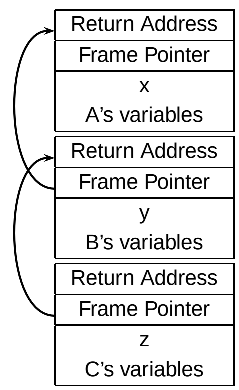
Names, Objects, and Bindings



Activation Records



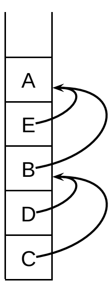
Activation Records




```
int A() {
  int x;
  B();
}

int B() {
  int y;
  C();
}

int C() {
  int z;
}
```

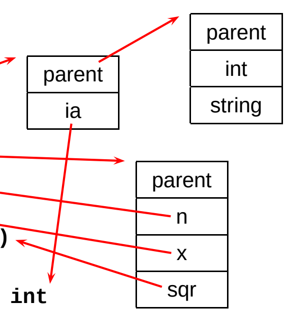
Nested Subroutines in Pascal

```
procedure A;
  procedure B;
    procedure C;
    begin .. end

    procedure D;
    begin C end
  begin D end

  procedure E;
  begin B end
begin E end
```



Symbol Tables in Tiger

```
let
  var n := 8
  var x := 3
  function sqr(a:int)
      = a * a
  type ia = array of int
in
  n := sqr(x)
end
```



Shallow vs. Deep binding

```
typedef int (*ifunc)();
ifunc foo() {
  int a = 1;
  int bar() { return a; }
  return bar;
}
int main() {
  ifunc f = foo();
  int a = 2;
  return (*f)();
}
```

	static	dynamic
shallow	1	2
deep	1	1

Shallow vs. Deep binding

```
void a(int i, void (*p)()) {

  void b() { printf("%d", i); }

  if (i=1) a(2,b) else (*p)();
}

void q() {}

int main() {
  a(1,q);
}
```

main()
a(1,q) i = 1, p = q b reference
a(2,b) i = 2, p = b
b

	static
shallow	2
deep	1

Static Semantic Analysis

Static Semantic Analysis

Lexical analysis: Make sure tokens are valid

```
if i 3 "This"              /* valid */
#a1123                     /* invalid */
```

Syntactic analysis: Makes sure tokens appear in correct order

```
for i := 1 to 5 do 1 + break /* valid */
if i 3                     /* invalid */
```

Semantic analysis: Makes sure program is consistent

```
let v := 3 in v + 8 end      /* valid */
let v := "f" in v(3) + v end /* invalid */
```

Static Semantic Analysis

Basic paradigm: recursively check AST nodes.

```
1 + break           1 - 5

  +                   -
 / \                 / \
1   break           1   5
```

check(+)
  check(1) = int
  check(break) = void
  FAIL: int ≠ void

check(-)
  check(1) = int
  check(5) = int
  Types match, return int

Ask yourself: at a particular node type, what must be true?

Mid-test Loops

```
while true do begin
  readln(line);
  if all_blanks(line) then goto 100;
  consume_line(line);
end;
100:
```

```
LOOP
  line := ReadLine;
WHEN AllBlanks(line) EXIT;
  ConsumeLine(line)
END;
```

Implementing multi-way branches

```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

Obvious way:

```
if (s == 1) { one(); }
else if (s == 2) { two(); }
else if (s == 3) { three(); }
else if (s == 4) { four(); }
```

Reasonable, but we can sometimes do better.

Implementing multi-way branches

If the cases are *dense*, a branch table is more efficient:

```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

```
labels l[] = { L1, L2, L3, L4 }; /* Array of labels */
if (s>=1 && s<=4) goto l[s-1];   /* not legal C */
L1: one(); goto Break;
L2: two(); goto Break;
L3: three(); goto Break;
L4: four(); goto Break;
Break:
```

Applicative- and Normal-Order Evaluation

```
int p(int i) { printf("%d ", i); return i; }

void q(int a, int b, int c)
{
  int total = a;
  printf("%d ", b);
  total += c;
}
```

What is printed by

```
q( p(1), 2, p(3) );
```

Applicative- and Normal-Order Evaluation

```
int p(int i) { printf("%d ", i); return i; }
void q(int a, int b, int c)
{
  int total = a;
  printf("%d ", b);
  total += c;
}
q( p(1), 2, p(3) );
```

Applicative: arguments evaluated before function is called.

Result: 1 3 2

Normal: arguments evaluated when used.

Result: 1 2 3

Applicative- vs. and Normal-Order

Most languages use applicative order.

Macro-like languages often use normal order.

```
#define p(x) (printf("%d ",x), x)
#define q(a,b,c) total = (a), \
   printf("%d ", (b)), \
   total += (c)

q( p(1), 2, p(3) );
```

Prints 1 2 3.

Some functional languages also use normal order evaluation to avoid doing work. “Lazy Evaluation”

Nondeterminism

Nondeterminism is not the same as random:

Compiler usually chooses an order when generating code.

Optimization, exact expressions, or run-time values may affect behavior.

Bottom line: don’t know what code will do, but often know set of possibilities.

```
int p(int i) { printf("%d ", i); return i; }
int q(int a, int b, int c) {}
q( p(1), p(2), p(3) );
```

Will *not* print 5 6 7. It will print one of

1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1