Review for the Midterm

Stephen A. Edwards

Columbia University

Fall 2014

The Midterm

75 minutes

Closed book

One double-sided sheet of notes of your own devising

Anything discussed in class is fair game

Little, if any, programming

Details of OCaml/C/C++/Java syntax not required

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

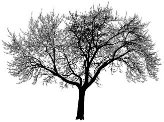
Lexical Analysis Gives Tokens

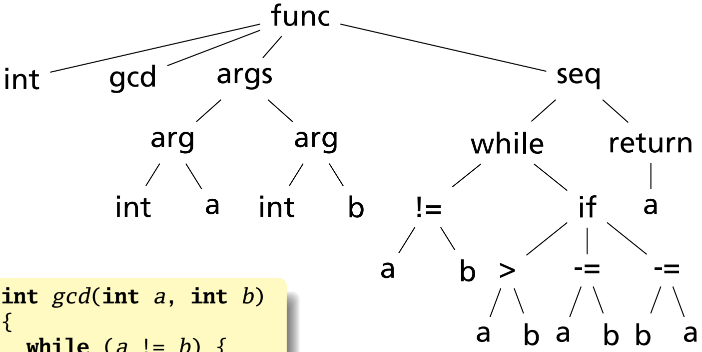
```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

`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.

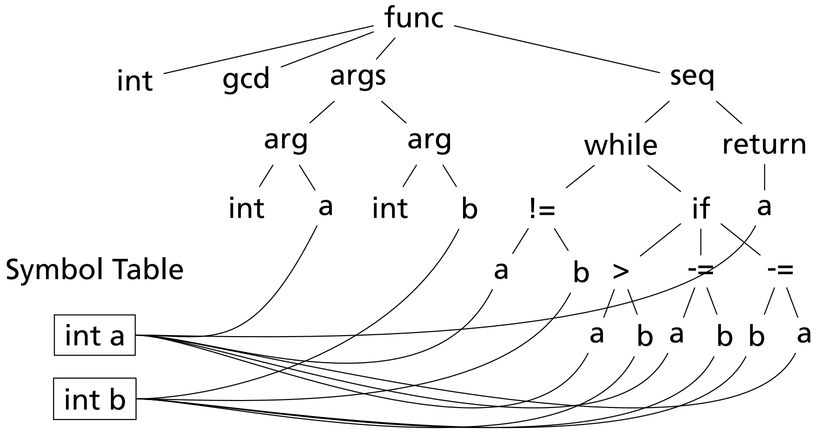
Parsing Gives an Abstract Syntax Tree






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

Semantic Analysis Resolves Symbols and Checks Types



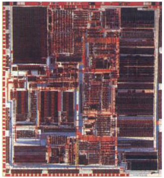
Translation into 3-Address Code

```
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 BP
       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
```

Describing Tokens

Alphabet: A finite set of symbols

Examples: $\{ 0, 1 \}$, $\{ A, B, C, \ldots, Z \}$, 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 $L = \{ \epsilon,$ wo $\}$, $M = \{$ man, men $\}$

Concatenation: Strings from one followed by the other

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

Union: All strings from each language

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

Kleene Closure: Zero or more concatenations

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

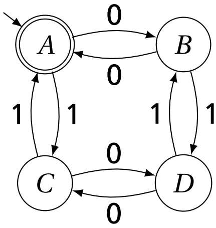
Regular Expressions over an Alphabet Σ

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 $L(s)$,
   - $(r) \mid (s)$ denotes $L(r) \cup L(s)$
   - $(r)(s)$ denotes $\{tu : t \in L(r), u \in L(s)\}$
   - $(r)^*$ denotes $\cup_{i=0}^{\infty} L^i$ ($L^0 = \{\epsilon\}$ and $L^i = LL^{i-1}$)

Nondeterministic Finite Automata

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



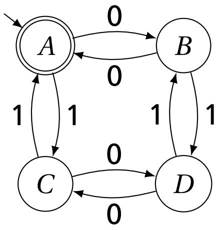
1. Set of states
   $S : \{ A \quad B \quad C \quad D \}$
2. Set of input symbols $\Sigma : \{0, 1\}$
3. Transition function $\sigma : S \times \Sigma_\epsilon \to 2^S$

state	ϵ	0	1
A	$\emptyset$	$\{B\}$	$\{C\}$
B	$\emptyset$	$\{A\}$	$\{D\}$
C	$\emptyset$	$\{D\}$	$\{A\}$
D	$\emptyset$	$\{C\}$	$\{B\}$

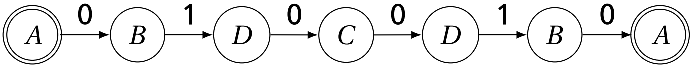
4. Start state $s_0 : A$
5. Set of accepting states
   $F : \{ A \}$

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.

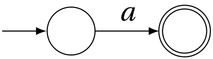


Show that the string "010010" is accepted.



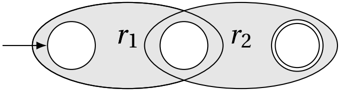
Translating REs into NFAs

a



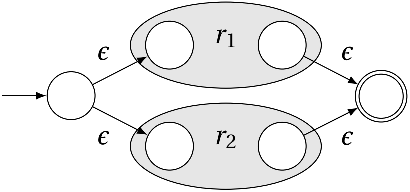
Symbol

$r_1 r_2$



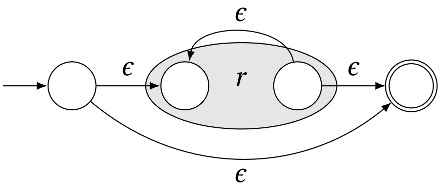
Sequence

$r_1 \mid r_2$



Choice

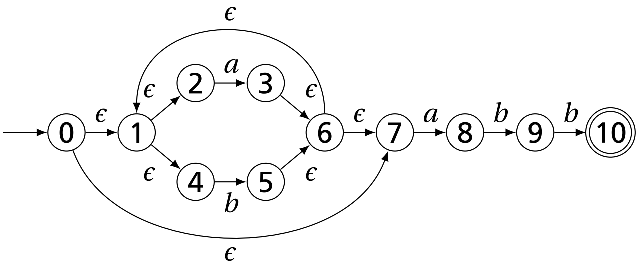
$(r)^*$



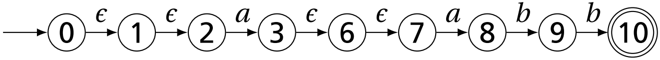
Kleene Closure

Translating REs into NFAs

Example: Translate $(a \mid b)^* abb$ into an NFA. Answer:



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



Simulating NFAs

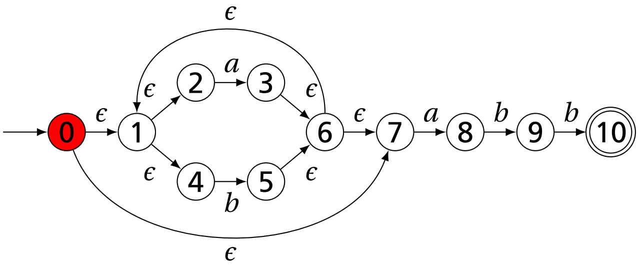
Problem: you must follow the "right" arcs to show that a string is accepted. How do you know which arc is right?

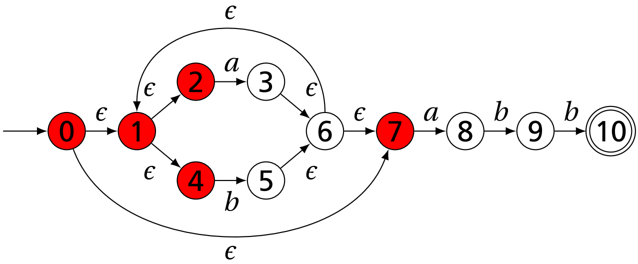
Solution: follow them all and sort it out later.

"Two-stack" NFA simulation algorithm:

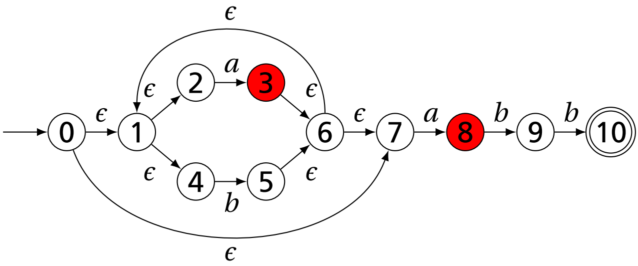
1. Initial states: the ϵ-closure of the start state
2. For each character c,
   - New states: follow all transitions labeled c
   - Form the ϵ-closure of the current states
3. Accept if any final state is accepting

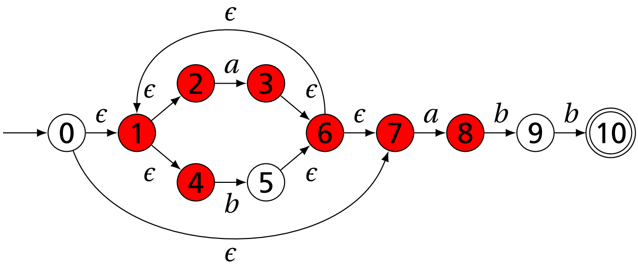
Simulating an NFA: $\cdot aabb$, Start



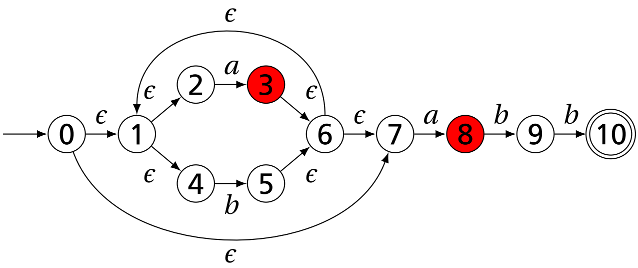


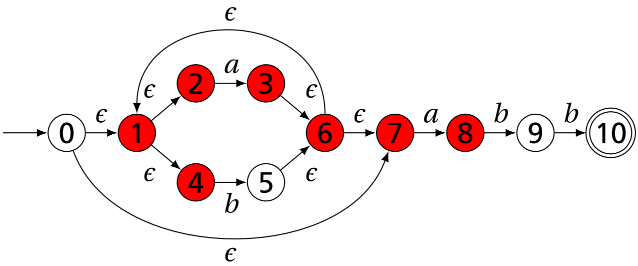
Simulating an NFA: $a{\cdot}abb$



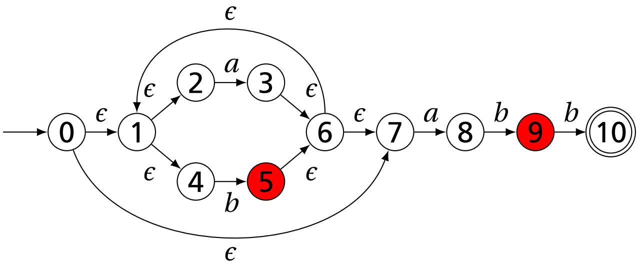


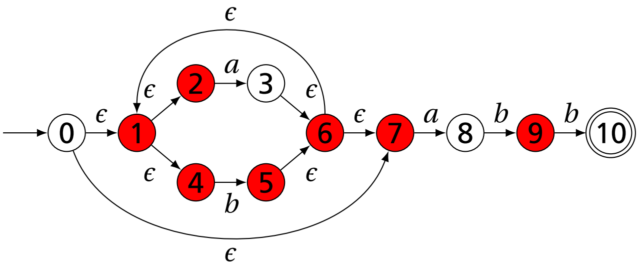
Simulating an NFA: $aa{\cdot}bb$



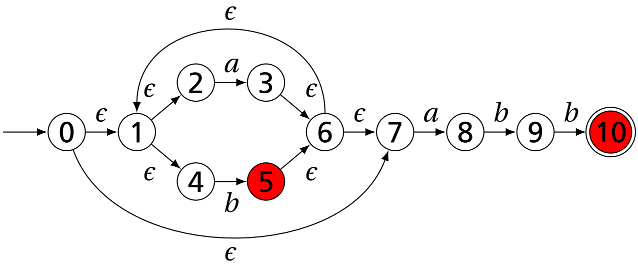


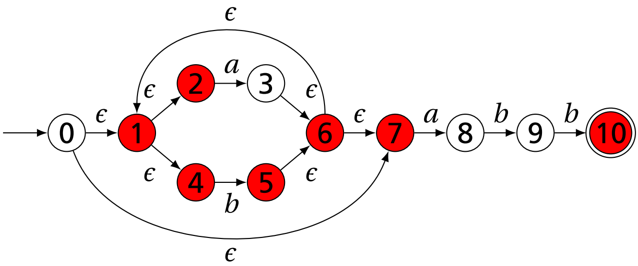
Simulating an NFA: $aab \cdot b$





Simulating an NFA: $aabb\cdot$, Done





Deterministic Finite Automata

Restricted form of NFAs:

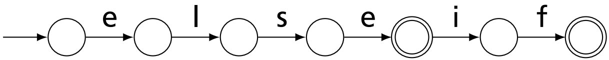
- No state has a transition on ϵ
- For each state s and symbol a, there is at most one edge labeled a leaving s.

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.

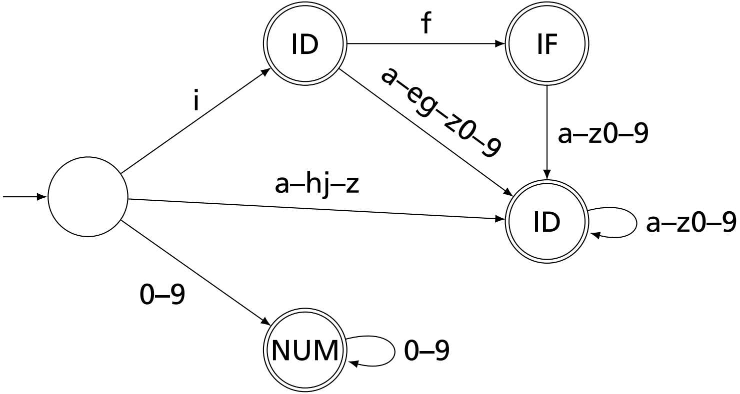
Deterministic Finite Automata

```
{
   type token = ELSE | ELSEIF
}

rule token =
  parse "else"    { ELSE }
      | "elseif" { ELSEIF }
```



Deterministic Finite Automata

```
{ type token = IF | ID of string | NUM of string }

rule token =
  parse "if"                                          { IF }
      | ['a'-'z'] ['a'-'z' '0'-'9']* as lit { ID(lit) }
      | ['0'-'9']+                          as num { NUM(num) }
```



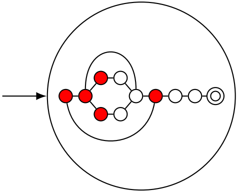
Building a DFA from an NFA

Subset construction algorithm

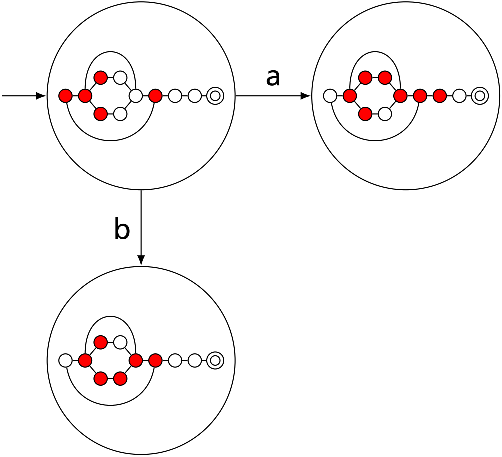
Simulate the NFA for all possible inputs and track the states that appear.

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

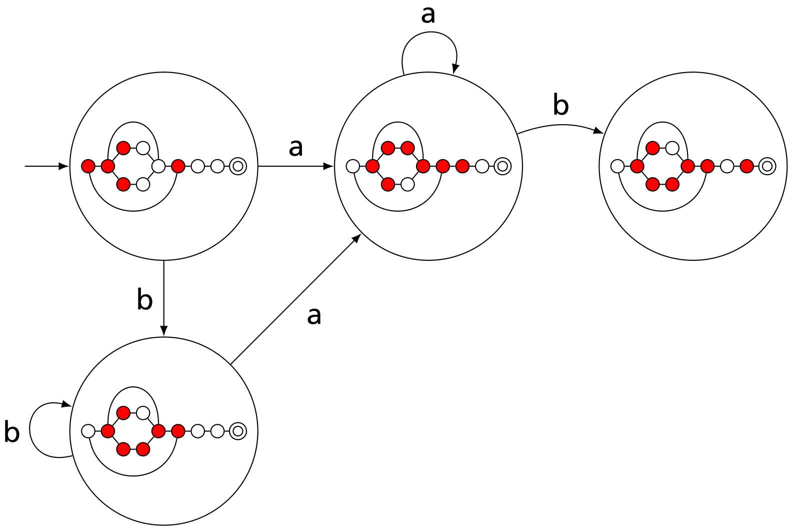
Subset construction for $(a \mid b)^* abb$



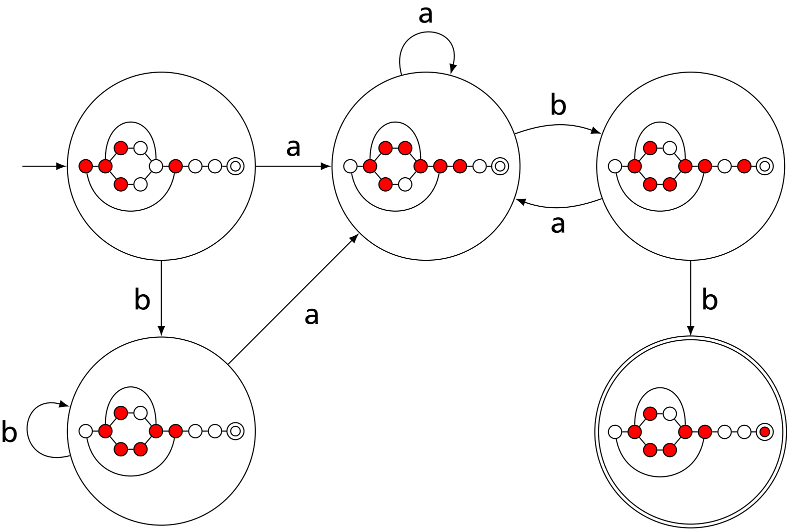
Subset construction for $(a \mid b)^* abb$



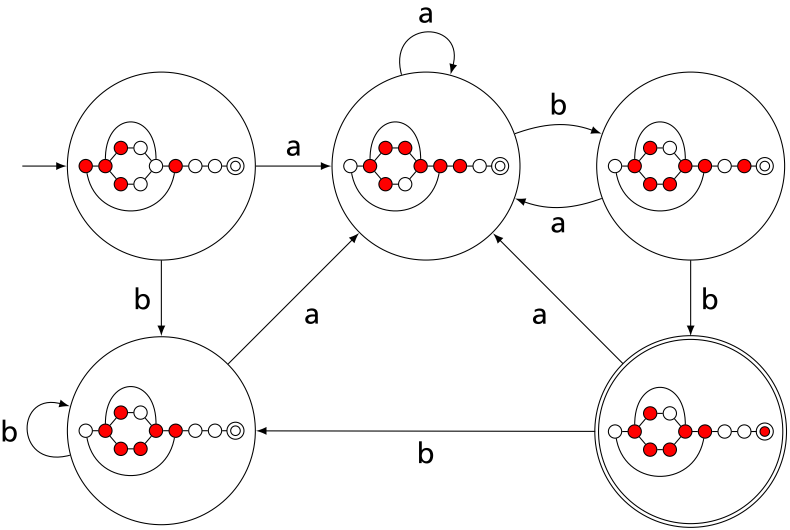
Subset construction for $(a \mid b)^* abb$



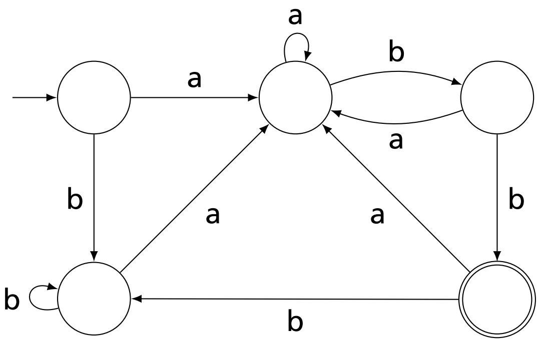
Subset construction for $(a \mid b)^* abb$



Subset construction for $(a \mid b)^* abb$



Result of subset construction for $(a \mid b)^* abb$



Is this minimal?

Ambiguous Arithmetic

Ambiguity can be a problem in expressions. Consider parsing

```
3 - 4 * 2 + 5
```

with the grammar

$$e \rightarrow e + e \mid e - e \mid e * e \mid e \,/\, e \mid N$$

```
      +          -                 *                -              -
     / \        / \              /   \             / \            / \
    -   5      3   +            -     +           3   *          *   5
   / \            / \          / \   / \             / \        / \
  3   *          *   5        3   4 2   5           4   +      +   2
     / \        / \                                    / \    / \
    4   2      4   2                                  2   5  3   4
```

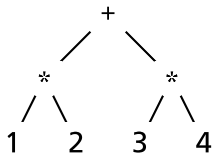
Operator Precedence

Defines how “sticky” an operator is.

`1 * 2 + 3 * 4`

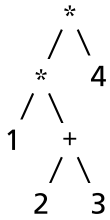
`*` at higher precedence than `+`:

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



`+` at higher precedence than `*`:

`1 * (2 + 3) * 4`



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
```

$((1-2)-3)-4$ $\quad\quad$ $1-(2-(3-4))$

left associative $\quad$ right associative

Fixing Ambiguous Grammars

A grammar specification:

```
expr :
    expr PLUS expr
  | expr MINUS expr
  | expr TIMES expr
  | expr DIVIDE expr
  | NUMBER
```

Ambiguous: no precedence or associativity.

Ocamlyacc's complaint: "16 shift/reduce conflicts."

Assigning Precedence Levels

Split into multiple rules, one per level

```
expr : expr PLUS expr
     | expr MINUS expr
     | term

term : term TIMES term
     | term DIVIDE term
     | atom

atom  : NUMBER
```

Still ambiguous: associativity not defined

Ocamlyacc's complaint: "8 shift/reduce conflicts."

Assigning Associativity

Make one side the next level of precedence

```
expr : expr PLUS term
     | expr MINUS term
     | term

term : term TIMES atom
     | term DIVIDE atom
     | atom

atom  : NUMBER
```

This is left-associative.

No shift/reduce conflicts.

Rightmost Derivation of $\mathbf{Id} * \mathbf{Id} + \mathbf{Id}$

$$
\begin{aligned}
1 &: e \rightarrow t + e \\
2 &: e \rightarrow t \\
3 &: t \rightarrow \mathbf{Id} * t \\
4 &: t \rightarrow \mathbf{Id}
\end{aligned}
$$

$$
\begin{gathered}
e \\
\underline{t + e} \\
t + \underline{t} \\
t + \underline{\mathbf{Id}} \\
\underline{\mathbf{Id} * t} + \mathbf{Id} \\
\mathbf{Id} * \underline{\mathbf{Id}} + \mathbf{Id}
\end{gathered}
$$

At each step, expand the *rightmost* nonterminal.

$$\frac{\text{nonterminal}}{\text{“handle”: The right side of a production}}$$

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambigious.

Rightmost Derivation: What to Expand

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

$$
\begin{array}{c}
e \\
\underline{t + e} \\
t + \underline{t} \\
t + \underline{\textbf{Id}} \\
\underline{\textbf{Id} * t} + \textbf{Id} \\
\textbf{Id} * \underline{\textbf{Id}} + \textbf{Id}
\end{array}
$$

$$
\begin{array}{r|l}
e & \\
\underline{t + e} & \\
t + \underline{t} & \\
t & + \underline{\textbf{Id}} \\
\underline{\textbf{Id} * t} & + \textbf{Id} \\
 & \textbf{Id} * \underline{\textbf{Id}} + \textbf{Id}
\end{array}
$$

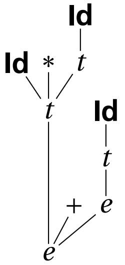
Expand here ↑ Terminals only

Reverse Rightmost Derivation

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

$$
\begin{array}{c}
e \\
\underline{t + e} \\
t + \underline{t} \\
t + \underline{\textbf{Id}} \\
\underline{\textbf{Id} * t} + \textbf{Id} \\
\textbf{Id} * \underline{\textbf{Id}} + \textbf{Id}
\end{array}
$$

viable prefixes	terminals
	$\textbf{Id} * \underline{\textbf{Id}} + \textbf{Id}$
$\underline{\textbf{Id} * t}$	$+ \textbf{Id}$
t	$+ \underline{\textbf{Id}}$
$t + \underline{t}$	
$\underline{t + e}$	
e	



Shift/Reduce Parsing Using an Oracle

$1: e \rightarrow t + e$

$2: e \rightarrow t$

$3: t \rightarrow \textbf{Id} * t$

$4: t \rightarrow \textbf{Id}$

$$
\begin{array}{c}
e \\
\underline{t + e} \\
t + \underline{t} \\
t + \underline{\textbf{Id}} \\
\underline{\textbf{Id} * t} + \textbf{Id} \\
\textbf{Id} * \underline{\textbf{Id}} + \textbf{Id}
\end{array}
$$

stack	input	
	$\textbf{Id} * \textbf{Id} + \textbf{Id}$	shift
$\textbf{Id}$	$* \textbf{Id} + \textbf{Id}$	shift
$\textbf{Id} *$	$\textbf{Id} + \textbf{Id}$	shift
$\textbf{Id} * \underline{\textbf{Id}}$	$+ \textbf{Id}$	reduce 4
$\underline{\textbf{Id} * t}$	$+ \textbf{Id}$	reduce 3
t	$+ \textbf{Id}$	shift
$t +$	$\textbf{Id}$	shift
$t + \underline{\textbf{Id}}$		reduce 4
$t + \underline{t}$		reduce 2
$\underline{t + e}$		reduce 1
e		accept

Handle Hunting

Right Sentential Form: any step in a rightmost derivation

Handle: in a sentential form, a RHS of a rule that, when rewritten, yields the previous step in a rightmost derivation.

The big question in shift/reduce parsing:

When is there a handle on the top of the stack?

Enumerate all the right-sentential forms and pattern-match against them? *Usually infinite in number, but let's try anyway.*

The Handle-Identifying Automaton

Magical result, due to Knuth: *An automaton suffices to locate a handle in a right-sentential form.*

$\mathbf{Id} * \mathbf{Id} * \cdots * \underline{\mathbf{Id} * t} \cdots$

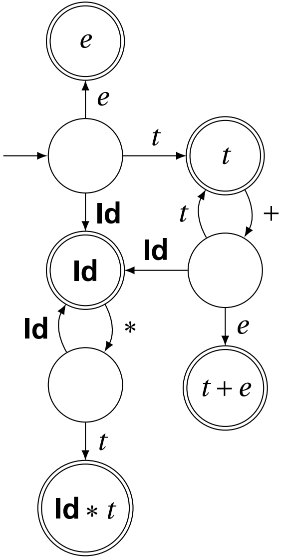
$\mathbf{Id} * \mathbf{Id} * \cdots * \underline{\mathbf{Id}} \cdots$

$t + t + \cdots + \underline{t + e}$

$t + t + \cdots + t + \underline{\mathbf{Id}}$

$t + t + \cdots + t + \mathbf{Id} * \mathbf{Id} * \cdots * \underline{\mathbf{Id} * t}$

$t + t + \cdots + \underline{t}$



Building the Initial State of the LR(0) Automaton

$$e' \rightarrow \cdot e$$

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

Key idea: automata identify viable prefixes of right sentential forms. Each state is an equivalence class of possible places in productions.

At the beginning, any viable prefix must be at the beginning of a string expanded from e. We write this condition "$e' \rightarrow \cdot e$"

Building the Initial State of the LR(0) Automaton

$$e' \rightarrow \cdot e$$
$$e \rightarrow \cdot t + e$$
$$e \rightarrow \cdot t$$

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

Key idea: automata identify viable prefixes of right sentential forms. Each state is an equivalence class of possible places in productions.

At the beginning, any viable prefix must be at the beginning of a string expanded from e. We write this condition "$e' \rightarrow \cdot e$"

There are two choices for what an e may expand to: $t + e$ and t. So when $e' \rightarrow \cdot e$, $e \rightarrow \cdot t + e$ and $e \rightarrow \cdot t$ are also true, i.e., it must start with a string expanded from t.

Building the Initial State of the LR(0) Automaton

$$
\begin{array}{|l|}\hline
e' \to \cdot e \\
e \to \cdot t + e \\
e \to \cdot t \\
t \to \cdot \mathbf{Id} * t \\
t \to \cdot \mathbf{Id} \\ \hline
\end{array}
$$

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

Key idea: automata identify viable prefixes of right sentential forms. Each state is an equivalence class of possible places in productions.

At the beginning, any viable prefix must be at the beginning of a string expanded from e. We write this condition "$e' \to \cdot e$"

There are two choices for what an e may expand to: $t + e$ and t. So when $e' \to \cdot e$, $e \to \cdot t + e$ and $e \to \cdot t$ are also true, i.e., it must start with a string expanded from t.

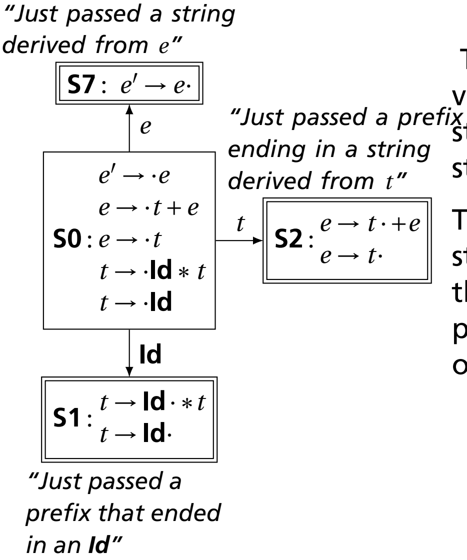
Similarly, t must be either $\mathbf{Id} * t$ or $\mathbf{Id}$, so $t \to \cdot \mathbf{Id} * t$ and $t \to \cdot \mathbf{Id}$.

Building the LR(0) Automaton

$$
\begin{array}{|l|}
\hline
\quad e' \to \cdot e \\
\quad e \to \cdot t + e \\
\mathbf{S0}: e \to \cdot t \\
\quad t \to \cdot \mathbf{Id} * t \\
\quad t \to \cdot \mathbf{Id} \\
\hline
\end{array}
$$

The first state suggests a viable prefix can start as any string derived from e, any string derived from t, or **Id**.

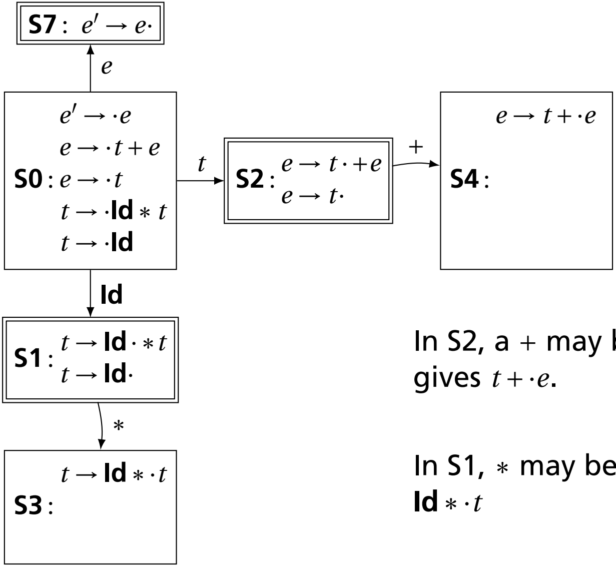
Building the LR(0) Automaton



The first state suggests a viable prefix can start as any string derived from e, any string derived from t, or **Id**.

The items for these three states come from advancing the $\cdot$ across each thing, then performing the closure operation (vacuous here).

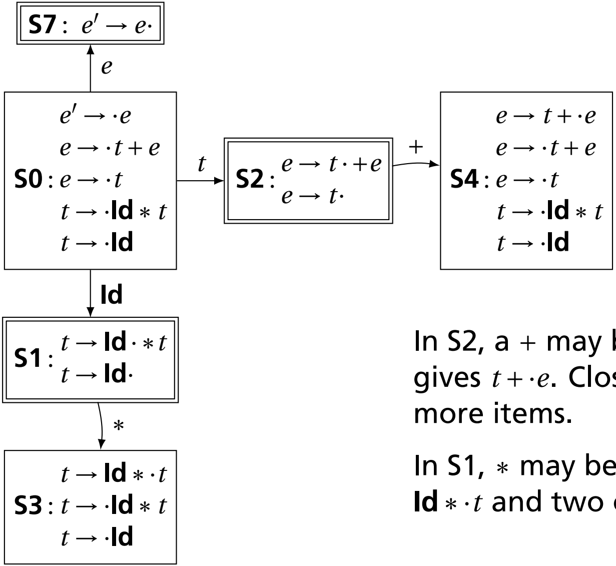
Building the LR(0) Automaton



In S2, a $+$ may be next. This gives $t + \cdot e$.

In S1, $*$ may be next, giving $\mathbf{Id} * \cdot t$

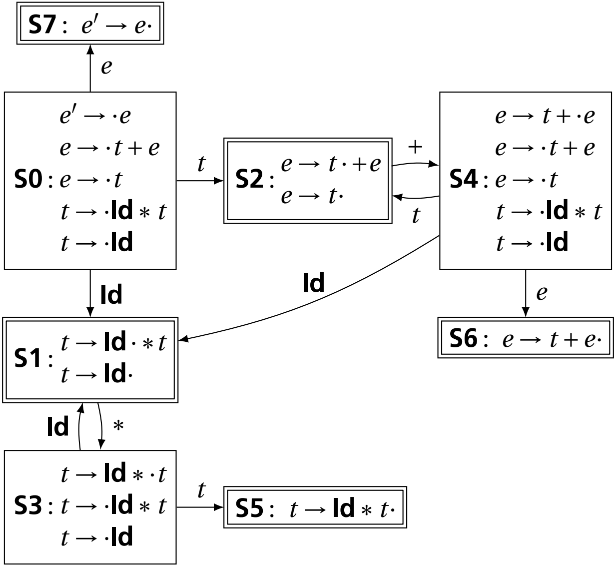
Building the LR(0) Automaton



In S2, a $+$ may be next. This gives $t + \cdot e$. Closure adds 4 more items.

In S1, $*$ may be next, giving **Id** $* \cdot t$ and two others.

Building the LR(0) Automaton



The first function

If you can derive a string that starts with terminal t from some sequence of terminals and nonterminals α, then $t \in \text{first}(\alpha)$.

1. Trivially, $\text{first}(X) = \{X\}$ if X is a terminal.
2. If $X \to \epsilon$, then add ϵ to $\text{first}(X)$.
3. For each production $X \to Y \cdots$, add $\text{first}(Y) - \{\epsilon\}$ to $\text{first}(X)$.
   If X can produce something, X can start with whatever that starts with
4. For each production $X \to Y_1 \cdots Y_k Z \cdots$ where $\epsilon \in \text{first}(Y_i)$ for $i = 1, \ldots, k$, add $\text{first}(Z) - \{\epsilon\}$ to $\text{first}(X)$.
   Skip all potential ϵ's at the beginning of whatever X produces

---

$1 : e \to t + e$

$2 : e \to t$

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

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

$\text{first}(\textbf{Id}) = \{\textbf{Id}\}$

$\text{first}(t) = \{\textbf{Id}\}$ because $t \to \textbf{Id} * t$ and $t \to \textbf{Id}$

$\text{first}(e) = \{\textbf{Id}\}$ because $e \to t + e$, $e \to t$, and $\text{first}(t) = \{\textbf{Id}\}$.

The follow function

If t is a terminal, A is a nonterminal, and $\cdots At \cdots$ can be derived, then $t \in$ follow(A).

1. Add \$ ("end-of-input") to follow(S) (start symbol).
   End-of-input comes after the start symbol
2. For each production $\rightarrow \cdots A\alpha$, add first$(\alpha) - \{\epsilon\}$ to follow(A).
   A is followed by the first thing after it
3. For each production $A \rightarrow \cdots B$ or $a \rightarrow \cdots B\alpha$ where $\epsilon \in$ first(α), then add everything in follow(A) to follow(B).
   If B appears at the end of a production, it can be followed by whatever follows that production

---

$1: e \rightarrow t + e$

$2: e \rightarrow t$

$3: t \rightarrow$ **Id** $* \; t$

$4: t \rightarrow$ **Id**

first$(t) = \{$**Id**$\}$

first$(e) = \{$**Id**$\}$

follow$(e) = \{\$\}$

follow$(t) = \{ \quad \}$

1. Because e is the start symbol

The follow function

If t is a terminal, A is a nonterminal, and $\cdots At \cdots$ can be derived, then $t \in$ follow(A).

1. Add \$ ("end-of-input") to follow(S) (start symbol).
   End-of-input comes after the start symbol
2. For each production $\rightarrow \cdots A\alpha$, add first$(\alpha) - \{\epsilon\}$ to follow(A).
   A is followed by the first thing after it
3. For each production $A \rightarrow \cdots B$ or $a \rightarrow \cdots B\alpha$ where $\epsilon \in$ first(α), then add everything in follow(A) to follow(B).
   If B appears at the end of a production, it can be followed by whatever follows that production

---

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

$3 : t \rightarrow$ **Id** $* \; t$

$4 : t \rightarrow$ **Id**

first$(t) = \{$**Id**$\}$

first$(e) = \{$**Id**$\}$

follow$(e) = \{\$\}$

follow$(t) = \{+ \quad \}$

2. Because $e \rightarrow \underline{t} + e$ *and* first$(+) = \{+\}$

The follow function

If t is a terminal, A is a nonterminal, and $\cdots At \cdots$ can be derived, then $t \in \text{follow}(A)$.

1. Add \$ ("end-of-input") to follow(S) (start symbol).
   End-of-input comes after the start symbol
2. For each production $\to \cdots A\alpha$, add first$(\alpha) - \{\epsilon\}$ to follow(A).
   A is followed by the first thing after it
3. For each production $A \to \cdots B$ or $a \to \cdots B\alpha$ where $\epsilon \in \text{first}(\alpha)$, then add everything in follow(A) to follow(B).
   If B appears at the end of a production, it can be followed by whatever follows that production

---

$1: e \to t + e$

$2: e \to t$

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

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

first$(t) = \{\textbf{Id}\}$

first$(e) = \{\textbf{Id}\}$

follow$(e) = \{\$\}$

follow$(t) = \{+, \$\}$

3. Because $e \to \underline{t}$ *and* $\$ \in \text{follow}(e)$

The follow function

If t is a terminal, A is a nonterminal, and $\cdots At \cdots$ can be derived, then $t \in$ follow(A).

1. Add \$ ("end-of-input") to follow(S) (start symbol).
   End-of-input comes after the start symbol
2. For each production $\rightarrow \cdots A\alpha$, add first$(\alpha) - \{\epsilon\}$ to follow(A).
   A is followed by the first thing after it
3. For each production $A \rightarrow \cdots B$ or $a \rightarrow \cdots B\alpha$ where $\epsilon \in$ first(α), then add everything in follow(A) to follow(B).
   If B appears at the end of a production, it can be followed by whatever follows that production

---

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

$3 : t \rightarrow \textbf{Id} \, * \, t$

$4 : t \rightarrow \textbf{Id}$

first$(t) = \{\textbf{Id}\}$

first$(e) = \{\textbf{Id}\}$

follow$(e) = \{\$\}$

follow$(t) = \{+, \$\}$

Fixed-point reached: applying any rule does not change any set

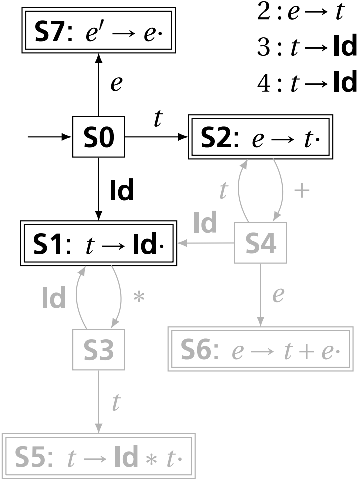
Converting the LR(0) Automaton to an SLR Parsing Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

$3 : t \rightarrow \textbf{Id} * t$

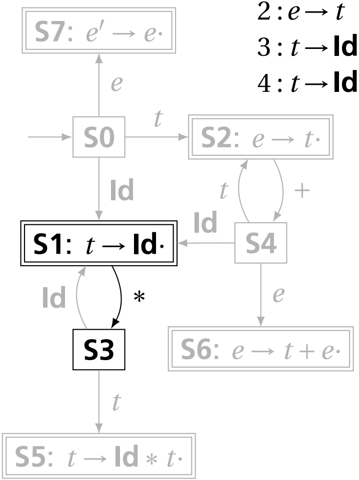
$4 : t \rightarrow \textbf{Id}$



State	Action				Goto	
	Id	$+$	$*$	$	e	t
0	s1				7	2

From S0, shift an **Id** and go to S1; or cross a t and go to S2; or cross an e and go to S7.

Converting the LR(0) Automaton to an SLR Parsing Table



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

State	Action				Goto	
	Id	$+$	$*$	$	e	t
0	s1				7	2
1		r4	s3	r4		

From S1, shift a $*$ and go to S3; or, if the next input could follow a t, reduce by rule 4. According to rule 1, $+$ could follow t; from rule 2, $ could.

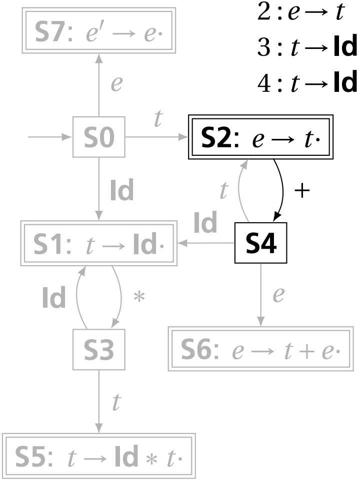
Converting the LR(0) Automaton to an SLR Parsing Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

$3: t \rightarrow \textbf{Id} * t$

$4: t \rightarrow \textbf{Id}$

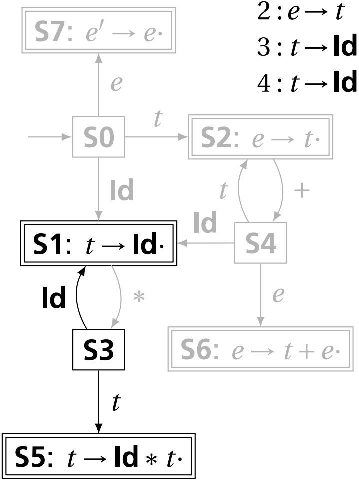


State	Action				Goto	
	Id	$+$	$*$	$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		

From S2, shift a $+$ and go to S4; or, if the next input could follow an e (only the end-of-input $), reduce by rule 2.

Converting the LR(0) Automaton to an SLR Parsing Table

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

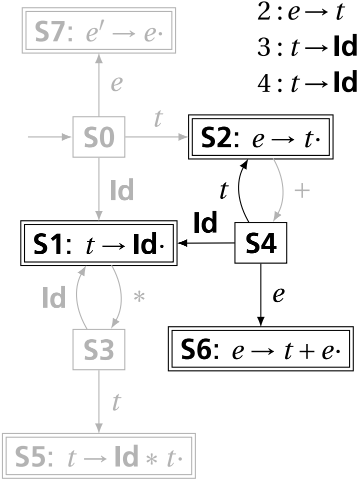


State	Action				Goto	
	Id	+	$*$	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5

From S3, shift an **Id** and go to S1; or cross a t and go to S5.

Converting the LR(0) Automaton to an SLR Parsing Table

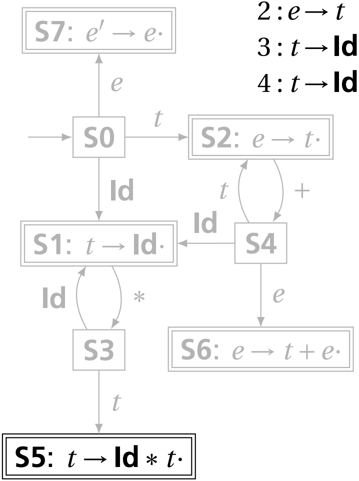
$1: e \rightarrow t + e$
$2: e \rightarrow t$
$3: t \rightarrow \textbf{Id} * t$
$4: t \rightarrow \textbf{Id}$



State	Action				Goto	
	Id	$+$	$*$	**$**	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2

From S4, shift an **Id** and go to S1; or cross an e or a t.

Converting the LR(0) Automaton to an SLR Parsing Table



$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

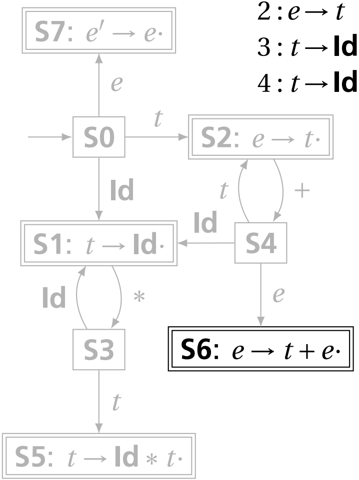
$3 : t \rightarrow$ **Id** $* \, t$

$4 : t \rightarrow$ **Id**

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

From S5, reduce using rule 3 if the next symbol could follow a t (again, $+$ and $).

Converting the LR(0) Automaton to an SLR Parsing Table

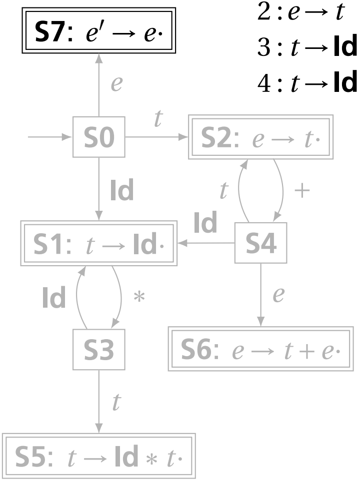


$1: e \to t + e$
$2: e \to t$
$3: t \to \textbf{Id} * t$
$4: t \to \textbf{Id}$

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

From S6, reduce using rule 1 if the next symbol could follow an e (\$ only).

Converting the LR(0) Automaton to an SLR Parsing Table



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

State	Action				Goto	
	Id	$+$	$*$	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

If, in S7, we just crossed an e, accept if we are at the end of the input.

Shift/Reduce Parsing with an SLR Table

$1: e \to t + e$

$2: e \to t$

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

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

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

Stack	Input	Action
0	**Id** * **Id** + **Id** $	Shift, goto 1

Look at the state on top of the stack and the next input token.

Find the action (shift, reduce, or error) in the table.

In this case, shift the token onto the stack and mark it with state 1.

Shift/Reduce Parsing with an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

$3 : t \rightarrow \mathbf{Id} * t$

$4 : t \rightarrow \mathbf{Id}$

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

Stack	Input	Action
0	**Id** $*$ **Id** $+$ **Id** $	Shift, goto 1
0 **Id** 1	$*$ **Id** $+$ **Id** $	Shift, goto 3

Here, the state is 1, the next symbol is $*$, so shift and mark it with state 3.

Shift/Reduce Parsing with an SLR Table

$1: e \rightarrow t + e$
$2: e \rightarrow t$
$3: t \rightarrow \mathbf{Id} * t$
$4: t \rightarrow \mathbf{Id}$

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

Stack	Input	Action
0	$\mathbf{Id} * \mathbf{Id} + \mathbf{Id}\,\$$	Shift, goto 1
0 Id/1	$* \mathbf{Id} + \mathbf{Id}\,\$$	Shift, goto 3
0 Id/1 $*$/3	$\mathbf{Id} + \mathbf{Id}\,\$$	Shift, goto 1
0 Id/1 $*$/3 Id/1	$+ \mathbf{Id}\,\$$	Reduce 4

Here, the state is 1, the next symbol is $+$. The table says reduce using rule 4.

Shift/Reduce Parsing with an SLR Table

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

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

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 4
0 **Id**/1 $*$/3	+ **Id** $	

Remove the RHS of the rule (here, just **Id**), observe the state on the top of the stack, and consult the "goto" portion of the table.

Shift/Reduce Parsing with an SLR Table

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

State	Action: Id	Action: +	Action: *	Action: $	Goto: e	Goto: t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

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 4
0 **Id** 1 $*$ 3 t 5	$+$ **Id** $	Reduce 3

Here, we push a t with state 5. This effectively "backs up" the LR(0) automaton and runs it over the newly added nonterminal.

In state 5 with an upcoming $+$, the action is "reduce 3."

Shift/Reduce Parsing with an SLR Table

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

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

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 4
0 **Id** 1 * 3 t 5	+ **Id** $	Reduce 3
0 t 2	+ **Id** $	Shift, goto 4

This time, we strip off the RHS for rule 3, **Id** $*$ t, exposing state 0, so we push a t with state 2.

Shift/Reduce Parsing with an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

$3 : t \rightarrow \textbf{Id} \; * \; t$

$4 : t \rightarrow \textbf{Id}$

State	Action: **Id**	Action: +	Action: ∗	Action: $	Goto: e	Goto: t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

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 4
0 **Id** 1 ∗ 3 t 5	+ **Id** $	Reduce 3
0 t 2	+ **Id** $	Shift, goto 4
0 t 2 + 4	**Id** $	Shift, goto 1
0 t 2 + 4 **Id** 1	$	Reduce 4
0 t 2 + 4 t 2	$	Reduce 2
0 t 2 + 4 e 6	$	Reduce 1
0 e 7	$	Accept

Types of Types

Type	Examples
Basic	Machine words, floating-point numbers, addresses/pointers
Aggregate	Arrays, structs, classes
Function	Function pointers, lambdas

Basic Types

Groups of data the processor is designed to operate on.

On an ARM processor,

Type	Width (bits)
Unsigned/two's-complement binary	
Byte	8
Halfword	16
Word	32
IEEE 754 Floating Point	
Single-Precision scalars & vectors	32, 64, .., 256
Double-Precision scalars & vectors	64, 128, 192, 256

Derived types

Array: a list of objects of the same type, often fixed-length

Record: a collection of named fields, often of different types

Pointer/References: a reference to another object

Function: a reference to a block of code

C’s Declarations and Declarators

Declaration: list of specifiers followed by a comma-separated list of declarators.

$$\underbrace{\texttt{static unsigned }\overbrace{\texttt{int}}^{\text{basic type}}}_{\text{specifiers}}\underbrace{\texttt{(*f[10])(int, char*);}}_{\text{declarator}}$$

Declarator’s notation matches that of an expression: use it to return the basic type.

Largely regarded as the worst syntactic aspect of C: both pre- (pointers) and post-fix operators (arrays, functions).

Structs

Structs are the precursors of objects:

Group and restrict what can be stored in an object, but not what operations they permit.

Can fake object-oriented programming:

```
struct poly { ... };

struct poly *poly_create();
void        poly_destroy(struct poly *p);
void        poly_draw(struct poly *p);
void        poly_move(struct poly *p, int x, int y);
int         poly_area(struct poly *p);
```

Unions: Variant Records

A `struct` holds all of its fields at once. A `union` holds only one of its fields at any time (the last written).

```
union token {
  int i;
  float f;
  char *string;
};

union token t;
t.i = 10;
t.f = 3.14159;       /* overwrite t.i */
char *s = t.string;  /* return gibberish */
```

Applications of Variant Records

A primitive form of polymorphism:

```
struct poly {
  int x, y;
  int type;
  union { int radius;
          int size;
          float angle; } d;
};
```

If `poly.type == CIRCLE`, use `poly.d.radius`.

If `poly.type == SQUARE`, use `poly.d.size`.

If `poly.type == LINE`, use `poly.d.angle`.

Name vs. Structural Equivalence

```
struct f {
  int x, y;
} foo = { 0, 1 };

struct b {
  int x, y;
} bar;

bar = foo;
```

Is this legal in C? Should it be?

Type Expressions

C's declarators are unusual: they always specify a name along with its type.

Languages more often have *type expressions*: a grammar for expressing a type.

Type expressions appear in three places in C:

```
(int *) a                          /* Type casts */
sizeof(float [10])                 /* Argument of sizeof() */
int f(int, char *, int (*)(int))   /* Function argument types */
```

Basic Static Scope in C, C++, Java, etc.

A name begins life where it is declared and ends at the end of its block.

From the CLRM, "The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block."

```
void foo()
{
   int x;



}
```

Hiding a Definition

Nested scopes can hide earlier definitions, giving a hole.

From the CLRM, “If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.”

```
void foo()
{
  int x;

  while ( a < 10 ) {
    int x;

  }


}
```

Static Scoping in Java

```
public void example() {
  // x, y, z not visible

  int x;
  // x visible

  for ( int y = 1 ; y < 10 ; y++ ) {
    // x, y visible

    int z;
    // x, y, z visible
  }

  // x visible
}
```

Basic Static Scope in O'Caml

A name is bound after the "in" clause of a "let." If the name is re-bound, the binding takes effect *after* the "in."

```
let x = 8 in


let x = x + 1 in
```

Returns the pair (12, 8):

```
let x = 8 in
 (let x =  x + 2 in
    x + 2),
x
```

Let Rec in O'Caml

The "rec" keyword makes a name visible to its definition. This only makes sense for functions.

```
let rec fib i =
  if i < 1 then 1 else
    fib (i-1) + fib (i-2)
in
  fib 5
```

```
(* Nonsensical *)
let rec x = x + 3 in
```

Let...and in O'Caml

```
let x = 8
and y = 9 in
```

Let...and lets you bind multiple names at once. Definitions are not mutually visible unless marked "rec."

```
let rec fac n =
     if n < 2 then
       1
     else
       n * fac1 n
and fac1 n = fac (n - 1)
in
fac 5
```

Nesting Function Definitions

```
let articles words =

  let report w =

    let count = List.length
      (List.filter ((=) w) words)
    in w ^ ": " ^
       string_of_int count

  in String.concat ", "
    (List.map report ["a"; "the"])

in articles
    ["the"; "plt"; "class"; "is";
     "a"; "pain"; "in";
     "the"; "butt"]
```

```
let count words w = List.length
  (List.filter ((=) w) words) in

let report words w = w ^ ": " ^
  string_of_int (count words w) in

let articles words =
  String.concat ", "
    (List.map (report words)
     ["a"; "the"]) in

articles
    ["the"; "plt"; "class"; "is";
     "a"; "pain"; "in";
     "the"; "butt"]
```

Produces “a: 1, the: 2”