Parser

Ronghui Gu
Spring 2019
Columbia University

* Course website: https://www.cs.columbia.edu/rgu/courses/4115/spring2019
** These slides are borrowed from Prof. Edwards.
The Big Picture
How do we describe/construct a program?
Solution: Use a Discrete Combinatorial System

Use combinations of a small number of things to represent (exponentially) many different things.
How do we combine characters into words?
int avg (int a, int b) ...

Lexical Analysis

Syntax Analysis

Semantic Analysis

Intermediate Code Generation

Optimization

Code Generation

0101110101...
How do we combine **words** into **sentences**?
int avg (int a, int b) ...

Lexical Analysis

Syntax Analysis

Semantic Analysis

Intermediate Code Generation

Optimization

Code Generation

0101110101...
| an integrated   | mobile       | network     |
| a parallel     | functional   | preprocessor|
| a virtual      | programmable | compiler    |
| an interactive | distributed  | system      |
| a responsive   | logical      | interface   |
| a synchronized | digital      | protocol    |
| a balanced     | concurrent   | architecture|
| a virtual      | knowledge-based | database   |
| a meta-level   | multimedia   | algorithm   |

E.g., “a responsive knowledge-based preprocessor.”

http://www.cs.purdue.edu/homes/dec/essay.topic.generator.html
How about more structured collections of things?

The boy eats hot dogs.
The dog eats ice cream.
Every happy girl eats candy.
A dog eats candy.
The happy happy dog eats hot dogs.

Pinker, *The Language Instinct*
If the boy eats hot dogs, then the girl eats ice cream.

Either the boy eats candy, or every dog eats candy.

Does this work?
Automata Have Poor Memories

Want to “remember” whether it is an “either-or” or “if-then” sentence. Only solution: duplicate states.
Problem: automata do not remember where they’ve been

\[
S \rightarrow \text{Either } A \\
S \rightarrow \text{If } A \\
A \rightarrow \text{the } B \\
A \rightarrow \text{the } C \\
A \rightarrow \text{a } B \\
A \rightarrow \text{a } C \\
A \rightarrow \text{every } B \\
A \rightarrow \text{every } C \\
B \rightarrow \text{happy } B \\
B \rightarrow \text{happy } C \\
C \rightarrow \text{boy } D \\
C \rightarrow \text{girl } D \\
C \rightarrow \text{dog } D \\
D \rightarrow \text{eats } E \\
E \rightarrow \text{hot dogs } F \\
E \rightarrow \text{ice cream } F \\
E \rightarrow \text{candy } F
\]
Context-Free Grammars have the ability to “call subroutines:”

\[
S \rightarrow \text{Either } P, \text{ or } P. \quad \text{Exactly two } P\text{s}
\]

\[
S \rightarrow \text{If } P, \text{ then } P.
\]

\[
P \rightarrow A \ H \ N \text{ eats } O \quad \text{One each of } A, \ H, \ N, \text{ and } O
\]

\[
A \rightarrow \text{the}
\]

\[
A \rightarrow a
\]

\[
A \rightarrow \text{every}
\]

\[
H \rightarrow \text{happy } H \quad H \text{ is “happy” zero or more times}
\]

\[
H \rightarrow \epsilon
\]

\[
N \rightarrow \text{boy}
\]

\[
N \rightarrow \text{girl}
\]

\[
N \rightarrow \text{dog}
\]

\[
O \rightarrow \text{hot dogs}
\]

\[
O \rightarrow \text{ice cream}
\]

\[
O \rightarrow \text{candy}
\]
An Example

$n$ 0’s followed by $n$ 1’s, e.g., 000111, 01

$S \rightarrow 0 \ S \ 1.$
$S \rightarrow \epsilon.$
Constructing Grammars and Ocamlyacc
Objective: build an abstract syntax tree (AST) for the token sequence from the scanner.

Goal: verify the syntax of the program, discard irrelevant information, and “understand” the structure of the program. Parentheses and most other forms of punctuation removed.
I shot an elephant in my pajamas

Jurafsky and Martin, *Speech and Language Processing*
The Dangling Else Problem

Who owns the else?

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

Problem comes after matching the first statement. Question is whether an “else” should be part of the current statement or a surrounding one since the second line tells us “stmt ELSE” is possible.
The Dangling Else Problem

Should this be

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

or

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

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

*As usual the “else” is resolved by connecting an else with the last encountered elseless if.*
The Dangling Else Problem

Idea: break into two types of statements: those that have a dangling “then” (“dstmt”) and those that do not (“cstmt”). A statement may be either, but the statement just before an “else” must not have a dangling clause because if it did, the “else” would belong to it.

```
stmt : dstmt
    | cstmt

dstmt : IF expr THEN stmt
     | IF expr THEN cstmt ELSE dstmt

cstmt : IF expr THEN cstmt ELSE cstmt
     | other statements...
```

```c
if (a) if (b) c(); else d();
```
The Dangling Else Problem

Idea: break into two types of statements: those that have a dangling “then” ("dstmt") and those that do not ("cstmt"). A statement may be either, but the statement just before an “else” must not have a dangling clause because if it did, the “else” would belong to it.

```
stmt : dstmt
     | cstmt

dstmt : IF expr THEN stmt
       | IF expr THEN cstmt ELSE dstmt

cstmt : IF expr THEN cstmt ELSE cstmt
       | other statements...
```

```
if (a) if (b) c(); else d();
cstmt?
```
We are effectively carrying an extra bit of information during parsing: whether there is an open “then” clause. Unfortunately, duplicating rules is the only way to do this in a context-free grammar.
Some languages resolve this problem by insisting on nesting everything.

E.g., Algol 68:

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

“fi” is “if” spelled backwards. The language also uses do–od and case–esac.
Ambiguity can be a problem in expressions. Consider parsing

\[ 3 - 4 \times 2 + 5 \]

with the grammar

\[ e \rightarrow e + e \mid e - e \mid e \times e \mid e / e \mid N \]
Operator Precedence and Associativity

Usually resolve ambiguity in arithmetic expressions
Like you were taught in elementary school:
“My Dear Aunt Sally”
Mnemonic for multiplication and division before addition and subtraction.
Operator Precedence

Defines how “sticky” an operator is.

\[ 1 \times 2 + 3 \times 4 \]

* at higher precedence than +:
\[ (1 \times 2) + (3 \times 4) \]

+ at higher precedence than *:
\[ 1 \times (2 + 3) \times 4 \]
Associativity

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

Most operators are left-associative

\[ 1 - 2 - 3 - 4 \]

\[ ((1 - 2) - 3) - 4 \]

left associative

\[ 1 - (2 - (3 - 4)) \]

right associative
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.”

\[1 \times 2 + 3?\]

expr TIMES expr PLUS shift?

expr TIMES expr PLUS reduce?
Assigning Precedence Levels

Split into multiple rules, one per level

\[
\begin{align*}
\text{expr} & : \text{expr} \ \text{PLUS} \ \text{expr} \\
& \quad | \ \text{expr} \ \text{MINUS} \ \text{expr} \\
& \quad | \ \text{term} \\
\text{term} & : \text{term} \ \text{TIMES} \ \text{term} \\
& \quad | \ \text{term} \ \text{DIVIDE} \ \text{term} \\
& \quad | \ \text{atom} \\
\text{atom} & : \text{NUMBER}
\end{align*}
\]

Still ambiguous: associativity not defined

Ocamlyacc’s complaint: “8 shift/reduce conflicts.”

\[
1 \times 2 + 3?
\]

\[
\text{term \ TIMES \ term \ PLUS} \ \text{cannot shift!}
\]

\[
\text{term \ TIMES \ term \ PLUS} \ \text{cannot reduce!}
\]

\[
\text{term \ TIMES \ term \ PLUS} \ \text{reduce!}
\]
Assigning Precedence Levels

Split into multiple rules, one per level

```plaintext
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.”

\[ 1 \times 2 \times 3? \]

term TIMES term TIMES shift?

term TIMES term PLUS reduce?
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.

```
1 * 2 * 3?
```

```
term TIMES atom TIMES cannot shift!
```

```
term TIMES atom TIMES cannot reduce!
```

```
term TIMES atom TIMES reduce!
```
Ocamlyacc Specifications

{%
(* Header: verbatim OCaml; optional *)
%

/* Declarations: tokens, precedence, etc. */

%

/* Rules: context-free rules */

%

(* Trailer: verbatim OCaml; optional *)

%/
Declarations

- %token `symbol` ...
  Define symbol names (exported to .mli file)
- %token `<type> symbol` ...
  Define symbols with attached attribute (also exported)
- %start `symbol` ...
  Define start symbols (entry points)
- %type `<type> symbol` ...
  Define the type for a symbol (mandatory for start)
- %left `symbol` ...
- %right `symbol` ...
- %nonassoc `symbol` ...
  Define precedence and associativity for the given symbols, listed in order from lowest to highest precedence
• `nonterminal` is the name of a rule, e.g., “program,” “expr”
• `symbol` is either a terminal (token) or another rule
• `semantic-action` is OCaml code evaluated when the rule is matched
• In a `semantic-action`, $1, 2, \ldots$ returns the value of the first, second, \ldots symbol matched
• A rule may include “%prec `symbol`” to override its default precedence
An Example .mly File

```ml
%token <int> INT
%token PLUS MINUS TIMES DIV LPAREN RPAREN EOL

%left PLUS MINUS /* lowest precedence */
%left TIMES DIV
%nonassoc UMINUS /* highest precedence */

%start main /* the entry point */
%type <int> main

main:
  expr EOL { $1 }

expr:
  INT { $1 }
  | LPAREN expr RPAREN { $2 }
  | expr PLUS expr { $1 + $3 }
  | expr MINUS expr { $1 - $3 }
  | expr TIMES expr { $1 * $3 }
  | expr DIV expr { $1 / $3 }
  | MINUS expr %prec UMINUS { - $2 }
```

/three.osf/four.osf
Parsing Algorithms
There are $O(n^3)$ algorithms for parsing arbitrary CFGs, but most compilers demand $O(n)$ algorithms.

Fortunately, the LL and LR subclasses of CFGs have $O(n)$ parsing algorithms. People use these in practice.
Rightmost Derivation of $\text{Id} \ast \text{Id} + \text{Id}$

\[ e \]

1. $e \rightarrow t + e$
2. $e \rightarrow t$
3. $t \rightarrow \text{Id} \ast t$
4. $t \rightarrow \text{Id}$

At each step, expand the *rightmost* nonterminal.

*nonterminal*

"handle": The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.
Rightmost Derivation of $\text{Id} \ast \text{Id} + \text{Id}$

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

At each step, expand the rightmost nonterminal.

“handle”: The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.
Rightmost Derivation of $\text{Id} \ast \text{Id} + \text{Id}$

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

At each step, expand the rightmost nonterminal.

“handle”: The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.
At each step, expand the rightmost nonterminal.

“handle”: The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.
Rightmost Derivation of \( \text{Id} \ast \text{Id} + \text{Id} \)

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

At each step, expand the rightmost nonterminal.

“handle”: The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.
Rightmost Derivation of \( \text{Id} \ast \text{Id} + \text{Id} \)

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

At each step, expand the rightmost nonterminal.

"handle": The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.
Rightmost Derivation of $\text{Id} \ast \text{Id} + \text{Id}$

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

At each step, expand the rightmost nonterminal.

"handle": The right side of a production

\[
\begin{align*}
e & \rightarrow t + e \rightarrow t + t \rightarrow t + \text{Id} \rightarrow \text{Id} \ast t + \text{Id} \rightarrow \text{Id} \ast \text{Id} + \text{Id}
\end{align*}
\]
Rightmost Derivation: What to Expand

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

Expand here ↑

Terminals only

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \textbf{Id} \ast t$
4: $t \rightarrow \textbf{Id}$
Reverse Rightmost Derivation

1: e → t + e
2: e → t
3: t → Id * t
4: t → Id

viable prefixes

Id * Id + Id

terminals
Reverse Rightmost Derivation

1: \( e \rightarrow t + e \)

2: \( e \rightarrow t \)

3: \( t \rightarrow \text{Id} \ast t \)

4: \( t \rightarrow \text{Id} \)

viable prefixes

terminals
Reverse Rightmost Derivation

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

viable prefixes | terminals
---|---
\( t + e \) | \( t + t \)
\( t + \text{Id} \) | \( \text{Id} \ast t + \text{Id} \)
\( \text{Id} \ast \text{Id} + \text{Id} \) | \( \text{Id} \ast t \)
Reverse Rightmost Derivation

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

viable prefixes  terminals

\[
\begin{align*}
& e \\
& t + e \\
& t + t \\
& t + \text{Id} \\
& \text{Id} \ast t + \text{Id} \\
& \text{Id} \ast \text{Id} + \text{Id}
\end{align*}
\]
Reverse Rightmost Derivation

1: \(e \rightarrow t + e\)
2: \(e \rightarrow t\)
3: \(t \rightarrow \text{Id} \ast t\)
4: \(t \rightarrow \text{Id}\)

Viable prefixes

Terminals
Reverse Rightmost Derivation

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

viable prefixes

terminals
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

Stack:

Input:
Shift/Reduce Parsing Using an Oracle

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

Stack:

Input:
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

Stack: 
Input: 

Shift/Reduce Parsing Rules: 
1. $e \rightarrow t + e$
2. $e \rightarrow t$
3. $t \rightarrow \text{Id} \ast t$
4. $t \rightarrow \text{Id}$

The diagram illustrates the parsing process with stack and input operations.
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow Id \ast t$
4: $t \rightarrow Id$

```
stack
Id \ast Id + Id
Id \ast Id + Id
Id \ast Id + Id
Id \ast Id + Id
Id \ast Id + Id

input
shift
shift
shift
reduce 4
```
1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

```
stack
input
```

```
Id \ast Id + Id
Id \ast Id + Id
Id \ast Id + Id
Id \ast Id + Id
Id \ast t + Id

Id \ast t + Id

Id \ast Id + Id
```

```
shift
shift
shift
reduce 4
reduce 3
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```

```
Id \ast Id + Id
```
1: \(e \rightarrow t + e\)
2: \(e \rightarrow t\)
3: \(t \rightarrow \text{Id} \ast t\)
4: \(t \rightarrow \text{Id}\)
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

```
stack
  Id \ast Id + Id
  Id \ast Id + Id
  Id \ast Id + Id
  Id \ast t + Id
  t + Id
input
  t + e
  t + t
  t + Id
```
Shift/Reduce Parsing Using an Oracle

1:  \( e \rightarrow t + e \)
2:  \( e \rightarrow t \)
3:  \( t \rightarrow Id \ast t \)
4:  \( t \rightarrow Id \)

[Diagram showing the parsing process with stack and input tokens marked with respective actions: shift, shift, reduce 4, reduce 3, shift, shift, reduce 4.]
1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} \ast t$

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

Stack:

```
Id \ast Id + Id
Id \ast Id + Id
Id \ast Id + Id
Id \ast t + Id
  t + Id
  t + t
  t + e
```

Input:

```
e

Shift
Shift
Shift
Reduce 4
Reduce 3
Shift
Shift
Reduce 4
Reduce 2
Reduce 1
```
Shift/Reduce Parsing Using an Oracle

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

Stack:
- \( e \)
- \( t + e \)
- \( t + t \)
- \( t + \text{Id} \ast t + \text{Id} \)
- \( \text{Id} \ast \text{Id} + \text{Id} \)

Input:
- \( e \ast t + \text{Id} \)
- \( t + \text{Id} \)
- \( t + t \)
- \( t + e \)

Actions:
- Shift
- Shift
- Shift
- Reduce 4
- Reduce 3
- Shift
- Shift
- Reduce 4
- Reduce 2
- Reduce 1
- 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 infinitely many; let’s try anyway.*
Some Right-Sentential Forms and Their Handles

1: e → t + e
2: e → t
3: t → Id * t
4: t → Id

```
           e
          /\  
         t   /
        /\  /
       Id * t Id
      /\  /\  
     Id * Id * t Id * Id
    /\  /\  /\  
   Id * Id * Id * t Id * Id * Id
```

Some Right-Sentential Forms and Their Handles

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

\[
\begin{array}{c}
\text{e} \\
\text{t} \\
\text{t} \\
\text{Id} \ast t \\
\text{Id} \ast t \\
\text{Id} \ast \text{Id} \ast t \\
\text{Id} \ast \text{Id} \ast \text{Id} \\
\text{Id} \ast \text{Id} \ast \text{Id} \ast \text{Id} \\
\text{Id} \ast \text{Id} \ast \text{Id} \ast \text{Id} \ast \text{Id} \\
\end{array}
\]

\[
\begin{array}{c}
t + e \\
\text{t} \\
\text{t} + \text{t} + \text{e} \\
\text{t} + \text{t} + \text{t} + \text{e} \\
\text{t} + \text{t} + \text{t} + \text{t} \\
\end{array}
\]
Some Right-Sentential Forms and Their Handles

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

Patterns:

- $\text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \cdots$
- $\text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \cdots$
- $t + t + \cdots + t + e$
- $t + t + \cdots + t + \text{Id}$
- $t + t + \cdots + t + \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t$
- $t + t + \cdots + t$
The Handle-Identifying Automaton

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

\[ \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \cdots \]
\[ \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \cdots \]
\[ t + t + \cdots + t + e \]
\[ t + t + \cdots + t + \text{Id} \]
\[ t + t + \cdots + t + \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \]
\[ t + t + \cdots + t \]
\[ e \]
Building the Initial State of the LR(o) Automaton

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

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 "\( e' \rightarrow \text{ce} \)"
Building the Initial State of the LR(o) Automaton

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

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 “\( e' \rightarrow \epsilon e \)”

There are two choices for what an \( e \) may expand to: \( t + e \) and \( t \). So when \( e' \rightarrow \epsilon e \), \( e \rightarrow \epsilon t + e \) and \( e \rightarrow \epsilon t \) are also true, i.e., it must start with a string expanded from \( t \).
Building the Initial State of the LR(0) Automaton

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 “$e' \rightarrow \epsilon e$”.

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

Also, $t$ must be $\text{id} \ast t$ or $\text{id}$, so $t \rightarrow \epsilon \text{id} \ast t$ and $t \rightarrow \epsilon \text{id}$.

This is a closure, like $\epsilon$-closure in subset construction.
Building the LR(0) Automaton

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e' \rightarrow \epsilon e$</td>
<td></td>
</tr>
<tr>
<td>$e \rightarrow \epsilon t + e$</td>
<td></td>
</tr>
</tbody>
</table>

So:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e \rightarrow \epsilon t$</td>
<td></td>
</tr>
<tr>
<td>$t \rightarrow \epsilon \text{Id} \ast t$</td>
<td></td>
</tr>
<tr>
<td>$t \rightarrow \epsilon \text{Id}$</td>
<td></td>
</tr>
</tbody>
</table>

The first state suggests a viable prefix can start as any string derived from $e$, any string derived from $t$, or $\text{Id}$.
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 \(\text{Id}\). The items for these three states come from advancing the \(\epsilon\) across each thing, then performing the closure operation (vacuous here).
Building the LR(0) Automaton

\[ S_7 : e' \rightarrow e \]
\[ S_0 : e \rightarrow t + e \]
\[ t \rightarrow Id * t \]
\[ t \rightarrow Id \]
\[ S_1 : t \rightarrow t*Id * t \]
\[ t \rightarrow Id \]
\[ * \]
\[ S_3 : t \rightarrow Id * t \]
\[ S_2 : e \rightarrow t + e \]
\[ e \rightarrow t \]
\[ S_4 : e \rightarrow t + e \]

In \( S_2 \), a + may be next. This gives \( t + e \).

In \( S_1 \), * may be next, giving \( Id * t \).
Building the LR(0) Automaton

**S7 :** $e' \rightarrow e$

**S0 :**
- $e' \rightarrow e$
- $e \rightarrow t + e$
- $t \rightarrow \text{Id} \ast t$
- $t \rightarrow \text{Id}$

**S2 :**
- $e \rightarrow t + e$
- $e \rightarrow t$

**S4 :**
- $e \rightarrow t + e$
- $e \rightarrow t$
- $t \rightarrow \text{Id} \ast t$
- $t \rightarrow \text{Id}$

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

In S1, $*$ may be next, giving $\text{Id} \ast e$ and two others.
Building the LR(0) Automaton

\(S_7 : \ e' \rightarrow e\)

\(S_0 : \ e \rightarrow t\)
  \(t \rightarrow \text{Id} \ast t\)
  \(t \rightarrow \text{Id}\)

\(S_1 : \ t \rightarrow \text{Id} \ast t\)
  \(\text{Id} \uparrow\)

\(S_3 : \ t \rightarrow \text{Id} \ast t\)
  \(t \rightarrow \text{Id}\)

\(S_2 : \ e \rightarrow t \ast e\)
  \(e \rightarrow t \ast e\)

\(S_4 : \ e \rightarrow t\)
  \(t \rightarrow \text{Id} \ast t\)
  \(t \rightarrow \text{Id}\)

\(S_5 : \ t \rightarrow \text{Id} \ast t\)

\(S_6 : \ e \rightarrow t + e\)

"Just passed a pre/f ix ending in a string derived from t"
"Just passed a pre/f ix that ended in an Id"
The /f_irst state suggests a viable pre/f ix 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 across each thing, then performing the closure operation (vacuous here).
In \(S_2\), a + may be next. This gives \(t + e\).
Closure adds \(S_4\) more items.
In \(S_6\), ∗ may be next, giving Id ∗ t and two others.
\(S_4 / S_6\)
### What to do in each state?

#### Stack

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \cdots )</td>
<td>( \ast \cdots )</td>
<td>Shift</td>
</tr>
<tr>
<td>( \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \cdots )</td>
<td>(+ \cdots)</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>( \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \cdots )</td>
<td></td>
<td>Syntax Error</td>
</tr>
</tbody>
</table>

#### Syntax Rules

1. \( e \rightarrow t + e \)
2. \( e \rightarrow t \)
3. \( t \rightarrow \text{Id} \ast t \)
4. \( t \rightarrow \text{Id} \)

#### Diagram

```
S1:  
  t → Id* t
  t → Id*

1: e → t + e
2: e → t
3: t → Id * t
4: t → Id
```

### Shift/Reduce Table

- \( \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \cdots \)
- \( t + t + \cdots + t + e \)
- \( t + t + \cdots + t + \text{Id} \)
- \( t + t + \cdots + t + \text{Id} \ast \text{Id} \ast \cdots \ast \text{Id} \ast t \)
- \( t + t + \cdots + t \)
- \( e \)
The FIRST function

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

1. If $X$ is a terminal, $\text{FIRST}(X) = \{X\}$.
2. If $X \rightarrow \epsilon$, then add $\epsilon$ to $\text{FIRST}(X)$.
3. If $X \rightarrow Y_1 \cdots Y_k$ and $\epsilon \in \text{FIRST}(Y_1)$, $\epsilon \in \text{FIRST}(Y_2)$, ..., and $\epsilon \in \text{FIRST}(Y_{i-1})$ for $i = 1, \ldots, k$ for some $k$,
   add $\text{FIRST}(Y_i) - \{\epsilon\}$ to $\text{FIRST}(X)$

   $X$ starts with anything that appears after skipping empty strings.

   Usually just $\text{FIRST}(Y_1) \subset \text{FIRST}(X)$

4. If $X \rightarrow Y_1 \cdots Y_K$ and $\epsilon \in \text{FIRST}(Y_1)$, $\epsilon \in \text{FIRST}(Y_2)$, ..., and $\epsilon \in \text{FIRST}(Y_k)$,
   add $\epsilon$ to $\text{FIRST}(X)$

   If all of $X$ can be empty, $X$ can be empty

\begin{align*}
1 : e & \rightarrow t + e & \text{FIRST}(\text{Id}) &= \{\text{Id}\} \\
2 : e & \rightarrow t & \text{FIRST}(t) &= \{\text{Id}\} \text{ because } t \rightarrow \text{Id} \ast t \text{ and } t \rightarrow \text{Id} \\
3 : t & \rightarrow \text{Id} \ast t & \text{FIRST}(e) &= \{\text{Id}\} \text{ because } e \rightarrow t + e, e \rightarrow t, \text{ and} \\
4 : t & \rightarrow \text{Id} & \text{FIRST}(t) &= \{\text{Id}\}.
\end{align*}
First and $\epsilon$

$\epsilon \in \text{FIRST}(\alpha)$ means $\alpha$ can derive the empty string.

1. If $X$ is a terminal, $\text{FIRST}(X) = \{X\}$.
2. If $X \rightarrow \epsilon$, then add $\epsilon$ to $\text{FIRST}(X)$.
3. If $X \rightarrow Y_1 \cdots Y_k$ and
   
   $\epsilon \in \text{FIRST}(Y_1)$, $\epsilon \in \text{FIRST}(Y_2)$, \ldots, and $\epsilon \in \text{FIRST}(Y_{i-1})$
   
   for $i = 1, \ldots, k$ for some $k$,
   
   add $\text{FIRST}(Y_i) - \{\epsilon\}$ to $\text{FIRST}(X)$
4. If $X \rightarrow Y_1 \cdots Y_K$ and
   
   $\epsilon \in \text{FIRST}(Y_1)$, $\epsilon \in \text{FIRST}(Y_2)$, \ldots, and $\epsilon \in \text{FIRST}(Y_k)$,
   
   add $\epsilon$ to $\text{FIRST}(X)$

\[
\begin{align*}
X & \rightarrow YZa \\
Y & \rightarrow \\
Y & \rightarrow b \\
Z & \rightarrow c \\
Z & \rightarrow W \\
W & \rightarrow \\
W & \rightarrow d
\end{align*}
\]
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 \( \text{FOLLOW}(S) \) (start symbol).

   \textit{End-of-input comes after the start symbol}

2. For each prod. \( \rightarrow \cdots A\alpha \), add \( \text{FIRST}(\alpha) - \{\epsilon\} \) to \( \text{FOLLOW}(A) \).

   \textit{A is followed by the first thing after it}

3. For each prod. \( A \rightarrow \cdots B \) or \( A \rightarrow \cdots B\alpha \) where \( \epsilon \in \text{FIRST}(\alpha) \), then add everything in \( \text{FOLLOW}(A) \) to \( \text{FOLLOW}(B) \).

   \textit{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: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

\( \text{FIRST}(t) = \{\text{Id}\} \)
\( \text{FIRST}(e) = \{\text{Id}\} \)

\( \text{FOLLOW}(e) = \{\$\} \)
\( \text{FOLLOW}(t) = \{\} \)

1. \textit{Because } \( e \) \textit{is the start symbol}
The FOLLOW function

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

1. Add $ ("end-of-input")$ to \( \text{FOLLOW}(S) \) (start symbol).

   *End-of-input comes after the start symbol*

2. For each prod. \( \rightarrow \cdots A\alpha \), add \( \text{FIRST}(\alpha) - \{ \epsilon \} \) to \( \text{FOLLOW}(A) \).

   *\( A \) is followed by the first thing after it*

3. For each prod. \( A \rightarrow \cdots B \) or \( A \rightarrow \cdots B\alpha \) where \( \epsilon \in \text{FIRST}(\alpha) \), then add everything in \( \text{FOLLOW}(A) \) to \( \text{FOLLOW}(B) \).

   *If \( B \) appears at the end of a production, it can be followed by whatever follows that production*

\[
1 : e \rightarrow t + e \\
2 : e \rightarrow t \\
3 : t \rightarrow \text{Id} \ast t \\
4 : t \rightarrow \text{Id} \\
\]

\( \text{FIRST}(t) = \{ \text{Id} \} \)

\( \text{FIRST}(e) = \{ \text{Id} \} \)

\( \text{FOLLOW}(e) = \{ \$ \} \)

\( \text{FOLLOW}(t) = \{ + \} \)

\( 2. \text{Because } e \rightarrow t + e \text{ and } \text{FIRST}(+) = \{ + \} \)
The FOLLOW function

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

1. Add $\$” (“end-of-input”) to $\text{FOLLOW}(S)$ (start symbol).
   
   *End-of-input comes after the start symbol*

2. For each prod. $\rightarrow \cdots A \alpha$, add $\text{FIRST}(\alpha) - \{\epsilon\}$ to $\text{FOLLOW}(A)$.
   
   *$A$ is followed by the first thing after it*

3. For each prod. $A \rightarrow \cdots B$ or $A \rightarrow \cdots B \alpha$ where $\epsilon \in \text{FIRST}(\alpha)$, then add everything in $\text{FOLLOW}(A)$ to $\text{FOLLOW}(B)$.
   
   *If $B$ appears at the end of a production, it can be followed by whatever follows that production*

\[
\begin{align*}
1: & \quad e \rightarrow t + e \\
2: & \quad e \rightarrow t \\
3: & \quad t \rightarrow \text{Id} \ast t \\
4: & \quad t \rightarrow \text{Id} \\
\text{FIRST}(t) & = \{\text{Id}\} \\
\text{FIRST}(e) & = \{\text{Id}\} \\
\text{FOLLOW}(e) & = \{\$\} \\
\text{FOLLOW}(t) & = \{+, \$\}
\end{align*}
\]

3. Because $e \rightarrow t$ 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 $t \in \text{FOLLOW}(A)$.

1. Add $\$$ (“end-of-input”) to $\text{FOLLOW}(S)$ (start symbol).
   
   *End-of-input comes after the start symbol*

2. For each prod. $\rightarrow \cdots A\alpha$, add $\text{FIRST}(\alpha) - \{\epsilon\}$ to $\text{FOLLOW}(A)$.
   
   *$A$ is followed by the first thing after it*

3. For each prod. $A \rightarrow \cdots B$ or $A \rightarrow \cdots B\alpha$ where $\epsilon \in \text{FIRST}(\alpha)$, then add everything in $\text{FOLLOW}(A)$ to $\text{FOLLOW}(B)$.

   *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: $t \rightarrow \text{Id} * t$

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

$\text{FIRST}(t) = \{\text{Id}\}$

$\text{FIRST}(e) = \{\text{Id}\}$

$\text{FOLLOW}(e) = \{\$$\}$

$\text{FOLLOW}(t) = \{+, \$$\}$

Fixed-point reached: applying any rule does not change any set
Converting the LR(o) Automaton to an SLR Table

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

FOLLOW(e) = {$$
FOLLOW(t) = \{+,$$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>+</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>7 2</td>
</tr>
</tbody>
</table>

1 : e → t + e
2 : e → t
3 : t → Id * t
4 : t → Id

S7: e' → e·
S0
S1: t → Id·
S2: e → t·
S3
S4: Id
S5: t → Id * t.
S6: e → t + e·
Converting the LR(0) Automaton to an SLR Table

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

**S7: \( e' \rightarrow e \).**

**S1: \( t \rightarrow \text{Id} \).**

**S2: \( e \rightarrow t \).**

**S3**

**S4:**

**S6: \( e \rightarrow t + e \).**

**S5: \( t \rightarrow \text{Id} \ast t \).**

**FOLLOW(e) = \{\$\}**

**FOLLOW(t) = \{+, \$\}**

**State** | **Action** | **Goto**
---|---|---
**Id** | + | $ \ |
| \( t \) | \( e \) | \( t \) |

| 0 | S1 | 7 2 |
| 1 | r4 s3 r4 |

From S1, shift a \( \ast \) and go to S3; or, if the next input \( \in \text{FOLLOW}(t) \), reduce by rule 4.
Converting the LR(0) Automaton to an SLR Table

1. \( e \rightarrow t + e \)
2. \( e \rightarrow t \)
3. \( t \rightarrow \text{Id} \ast t \)
4. \( t \rightarrow \text{Id} \)

FOLLOW(\( e \)) = \{\$\}
FOLLOW(\( t \)) = \{+, \$\}

From S2, shift a + and go to S4; or, if the next input \( \in \) FOLLOW(\( e \)), reduce by rule 2.
Converting the LR(o) Automaton to an SLR Table

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

**Follow**: $e (\text{Id} = \{\}$, $t (\text{Id} = \{+, $\}$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>r4</td>
<td>s3</td>
</tr>
<tr>
<td>2</td>
<td>S4</td>
<td>r2</td>
</tr>
<tr>
<td>3</td>
<td>S1</td>
<td>5</td>
</tr>
</tbody>
</table>

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 Table

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

**State** | **Action** | **Goto**
---|---|---
\text{Id} | + \ast $ | $ e \ t$
0 | S1 | 7 2
1 | r4  s3  r4 | 
2 | S4  r2 | 
3 | S1 | 5
4 | S1 | 6 2

FOLLOW($e$) = {$\$$}
FOLLOW($t$) = {+, $\$$}

From S4, shift an \text{Id} and go to S1; or cross an $e$ or a $t$. 
Converting the LR(0) Automaton to an SLR Table

### State Action Goto

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: e → t + e</td>
<td>+, *</td>
<td>$</td>
</tr>
<tr>
<td>2: e → t</td>
<td></td>
<td>s1</td>
</tr>
<tr>
<td>3: t → Id * t</td>
<td></td>
<td>r4, s3, r4</td>
</tr>
<tr>
<td>4: t → Id</td>
<td></td>
<td>s4</td>
</tr>
<tr>
<td>5: t → Id * t</td>
<td></td>
<td>s1</td>
</tr>
<tr>
<td>6: e → t + e</td>
<td></td>
<td>r3</td>
</tr>
</tbody>
</table>

FOLLOW(e) = {$}$
FOLLOW(t) = {+, $}$

From S5, reduce using rule 3 if the next symbol ∈ FOLLOW(t).
Converting the LR(0) Automaton to an SLR Table

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow \text{Id} \ast t \)
4: \( t \rightarrow \text{Id} \)

FOLLOW(\( e \)) = \{\$, \}
FOLLOW(\( t \)) = \{+, \$\}

- \( S_0 \)
  - \( S_2: e \rightarrow t \cdot \)
  - \( S_1: t \rightarrow \text{Id} \cdot \)
  - \( S_3: \text{Id} \cdot \)
  - \( S_4 \)
  - \( S_5: t \rightarrow \text{Id} \ast t \cdot \)
  - \( S_7: e' \rightarrow e \cdot \)

**State** | **Action** | **Goto**
---|---|---
\( \text{Id} \) | +, * | $ |
0 | \( s_1 \) | 7, 2 |
1 | r4, s3, r4 | |
2 | s4, r2 | |
3 | s1 | 5 |
4 | s1 | 6, 2 |
5 | r3, r3 | |
6 | r1 | |

From \( S_6 \), reduce using rule 1 if the next symbol \( \in \) FOLLOW(\( e \)).
Converting the LR(0) Automaton to an SLR Table

1: $e \rightarrow t + e$
2: $e \rightarrow t$
3: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

**S7: $e' \rightarrow e$.**

*State Action Goto*

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>+</td>
<td>$\ast$</td>
</tr>
<tr>
<td>Id</td>
<td>$\ast$</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>$+$</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>$e \rightarrow t$.</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>$t \rightarrow \text{Id}$.</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>$t \rightarrow \text{Id} \ast t$.</td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>$e \rightarrow t + e$.</td>
<td></td>
</tr>
</tbody>
</table>

**Follows:**

- FOLLOW($e$) = \{$\$\}
- FOLLOW($t$) = \{+$,$\}

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

\[ 1 : e \to t + e \]
\[ 2 : e \to t \]
\[ 3 : t \to \text{Id} \ast t \]
\[ 4 : t \to \text{Id} \]

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S1</td>
<td>e</td>
</tr>
<tr>
<td>1</td>
<td>r4</td>
<td>s3</td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td>r2</td>
</tr>
<tr>
<td>3</td>
<td>S1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\textbf{Id} \ast \textbf{Id} + \textbf{Id} $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0</td>
<td>$\textbf{Id}$</td>
<td>$\ast \textbf{Id} + \textbf{Id} $</td>
</tr>
</tbody>
</table>

Here, the state is 1, the next symbol is $\ast$, so shift and mark it with state 3.
1 : $e \rightarrow t + e$
2 : $e \rightarrow t$
3 : $t \rightarrow \text{Id} \ast t$
4 : $t \rightarrow \text{Id}$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Id * Id + Id $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>1</td>
<td>* Id + Id $</td>
<td>Shift, goto 3</td>
</tr>
<tr>
<td>2</td>
<td>Id + Id $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>3</td>
<td>+ Id $</td>
<td>Reduce 4</td>
</tr>
</tbody>
</table>

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: $t \rightarrow \text{Id} \times t$
4: $t \rightarrow \text{Id}$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>+</td>
<td>$$</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td>r2</td>
</tr>
<tr>
<td>3</td>
<td>S1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>S1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Remove the RHS of the rule (the handle: here, just \text{Id}), observe the state on the top of the stack, and consult the “goto” portion of the table.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Id * Id + Id $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>1</td>
<td>* Id + Id $</td>
<td>Shift, goto 3</td>
</tr>
<tr>
<td>0</td>
<td>Id + Id $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>1</td>
<td>+ Id $</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0</td>
<td>Id * Id</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>+ Id $</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Id</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Id * Id</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Id</td>
<td></td>
</tr>
</tbody>
</table>

52
### Shift/Reduce Parsing with an SLR Table

1: \( e \to t + e \)

2: \( e \to t \)

3: \( t \to \text{Id} * t \)

4: \( t \to \text{Id} \)

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Id} )</td>
<td>+ * $</td>
<td>( e \ t )</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>7  2</td>
</tr>
<tr>
<td>1</td>
<td>( r_4 \ s_3 \ r_4 )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( s_4 \ r_2 )</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>S1</td>
<td>6  2</td>
</tr>
<tr>
<td>4</td>
<td>S1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td>( r_3 )</td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \text{Id} * \text{Id} + \text{Id} $ )</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0</td>
<td>( * \text{Id} + \text{Id} $ )</td>
<td>Shift, goto 3</td>
</tr>
<tr>
<td>1</td>
<td>( \text{Id} + \text{Id} $ )</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0</td>
<td>( + \text{Id} $ )</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0</td>
<td>( + \text{Id} $ )</td>
<td>Reduce 3</td>
</tr>
</tbody>
</table>

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: $t \rightarrow \text{Id} \ast t$
4: $t \rightarrow \text{Id}$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Id</td>
<td>e</td>
</tr>
<tr>
<td>1</td>
<td>r4</td>
<td>s3</td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td>r2</td>
</tr>
<tr>
<td>3</td>
<td>S1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>S1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

This time, we strip off the RHS for rule 3, the handle $\text{Id} \ast t$, 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 : \( t \rightarrow \text{Id} \ast t \)
4 : \( t \rightarrow \text{Id} \)

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S1</td>
<td>e</td>
</tr>
<tr>
<td>1</td>
<td>r4</td>
<td>s3</td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td>r2</td>
</tr>
<tr>
<td>3</td>
<td>S1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>S1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Id</td>
<td>* Id + Id $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0 Id</td>
<td>* Id + Id $</td>
<td>Shift, goto 3</td>
</tr>
<tr>
<td>0 Id</td>
<td>Id + Id $</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0 Id</td>
<td>1 3 Id</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0 Id</td>
<td>1 3 Id</td>
<td>Reduce 3</td>
</tr>
<tr>
<td>0 Id</td>
<td>1 3 5 t</td>
<td>Shift, goto 4</td>
</tr>
<tr>
<td>0 t</td>
<td>t + Id</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0 t</td>
<td>t + Id</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0 t</td>
<td>t + t</td>
<td>Reduce 2</td>
</tr>
<tr>
<td>0 t</td>
<td>t + e</td>
<td>Reduce 1</td>
</tr>
<tr>
<td>0 t</td>
<td>e</td>
<td>Accept</td>
</tr>
<tr>
<td>0 t</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>0 t</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>0 t</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>
LR parser: “Bottom-up parser”:
L = Left-to-right scan, R = (reverse) Rightmost derivation

LL parser: “Top-down parser”:
L = Left-to-right scan: L = (reverse) Leftmost derivation

LR(1): LR parser that considers next token (lookahead of 1)
LR(0): Only considers stack to decide shift/reduce

SLR(1): Simple LR: lookahead from first/follow rules
Derived from LR(0) automaton

LALR(1): Lookahead LR(1): fancier lookahead analysis
Uses same LR(0) automaton as SLR(1)

Ocamlyacc builds LALR(1) tables.
This is a tricky, but mechanical procedure. The Ocamlyacc parser generator uses a modified version of this technique to generate fast bottom-up parsers.

You need to understand it to comprehend error messages:

**Shift/reduce conflicts** are caused by a state like

\[ t \rightarrow \cdot \textbf{Else } s \]
\[ t \rightarrow \cdot \]

If the next token is **Else**, do you reduce it since **Else** may follow a \( t \), or shift it?

**Reduce/reduce conflicts** are caused by a state like

\[ t \rightarrow \textbf{Id } \ast t \cdot \]
\[ e \rightarrow \textbf{Id } \ast t \cdot \]

Do you reduce by “\( t \rightarrow \textbf{Id } \ast t \)” or by “\( e \rightarrow \textbf{Id } \ast t \)”?
A Reduce/Reduce Conflict

\[
\begin{align*}
S_1 : & \quad a' \rightarrow a \\
S_2 : & \quad a \rightarrow \text{Id} \cdot \text{Id} \\
S_3 : & \quad a \rightarrow b \\
S_4 : & \quad a \rightarrow \text{Id} \cdot \text{Id} \\
\end{align*}
\]

1 : \( a \rightarrow \text{Id} \ \text{Id} \)
2 : \( a \rightarrow b \)
3 : \( b \rightarrow \text{Id} \ \text{Id} \)