Parsing

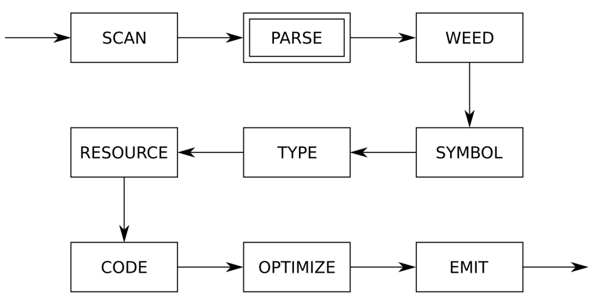
COMP 520: Compiler Design (4 credits)

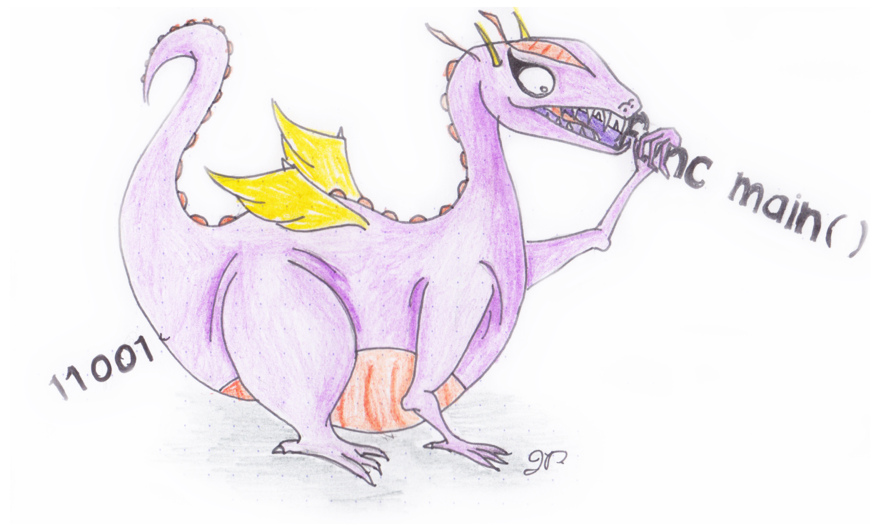
Alexander Krolik

`alexander.krolik@mail.mcgill.ca`

MWF 8:30-9:30, TR 1080

`http://www.cs.mcgill.ca/~cs520/2019/`





Readings

Crafting a Compiler (recommended)

- Chapter 4.1 to 4.4
- Chapter 5.1 to 5.2
- Chapter 6.1, 6.2 and 6.4

Crafting a Compiler (optional)

- Chapter 4.5
- Chapter 5.3 to 5.9
- Chapter 6.3 and 6.5

Modern Compiler Implementation in Java

- Chapter 3

Tool Documentation (links on `http://www.cs.mcgill.ca/~cs520/2019/`)

- flex, bison, and/or SableCC

Announcements (Monday, January 14th)

Milestones

- Continue picking your group (3 recommended). Who doesn't have a group?
- Learn `flex`/`bison` or `SableCC` – Assignment 1 out today!

Midterm

- 1.5 hour evening midterm, 6:00-7:30 PM
- **Date**: February 26 or 27 in McConnell 103/321. *Which is preferred?*

Office Hours

- **Monday/Wednesday**: 9:30-10:30
- If this does not work for you then please do send a message via email, Facebook group, etc.

Parsing

The parsing phase of a compiler

- Is the second phase of a compiler;
- Is also called syntactic analysis;
- Takes a string of tokens generated by the scanner as input; and
- Builds a *parse tree* using a *context-free grammar*.

Internally

- It corresponds to a *deterministic pushdown automaton*;
- Plus some glue code to make it work; and
- Can be generated by `bison` (or `yacc`), CUP, ANTLR, SableCC, Beaver, JavaCC, ...

Pushdown Automata

Regular languages (equivalently regexps/DFAs/NFAs) are not sufficient powerful to recognize some aspects of programming languages. A *pushdown automaton* is a more powerful tool that

- Is a FSM + an unbounded stack;
- The stack can be viewed/manipulated by transitions;
- Is used to recognize a context-free language;
- i.e. A larger set of languages to DFAs/NFAs.

Example: How can we recognize the language of matching parentheses using a PDA? (where the number of parentheses is unbounded)

$$\{\, (^{n}\,)^{n} \mid n \geq 1 \,\} = (),\ (()),\ ((())),\ \ldots$$

Key idea: We can use the stack for matching!

Context-Free Languages

A *context-free language* is a language derived from a *context-free grammar*

Context-Free Grammars

A *context-free grammar* is a 4-tuple (V, Σ, R, S), where

- V: set of *variables* (or *non-terminals*)
- Σ: set of *terminals* such that $V \cap \Sigma = \emptyset$
- R: set of *rules*, where the LHS is a variable in V and the RHS is a string of variables in V and terminals in Σ
- $S \in V$: start variable

Example Context-Free Grammar

A context-free grammar specifies rules of the form $A \rightarrow \gamma$ where A is a variable, and γ contains a sequence of terminals/non-terminals.

Simple CFG	**Alternatively**
$A \rightarrow$ a B	$A \rightarrow$ a B | ϵ
$A \rightarrow \epsilon$	$B \rightarrow$ b B | c
$B \rightarrow$ b B	
$B \rightarrow$ c	

In both cases we specify $S = A$

Language

This CFG generates either (a) the empty string; or (b) strings that

- Start with exactly 1 “a”; followed by zero or more “b”s; and end with 1 “c”.
- i.e. ϵ, `ac, abc, abbc, abbbc, ...`

Can you write this grammar as a regular expression?

Context-Free Grammars

In the language hierarchy, context-free grammars

- Are stronger than regular expressions;
- Generate context-free languages; and
- Are able to express some recursively-defined constructs not possible in regular expressions.

Example: Returning to the previous language for which we defined a PDA

$$\{ (^n)^n \mid n \geq 1 \} = (),\ (()),\ ((())),\ \ldots$$

The solution using a CFG is simple

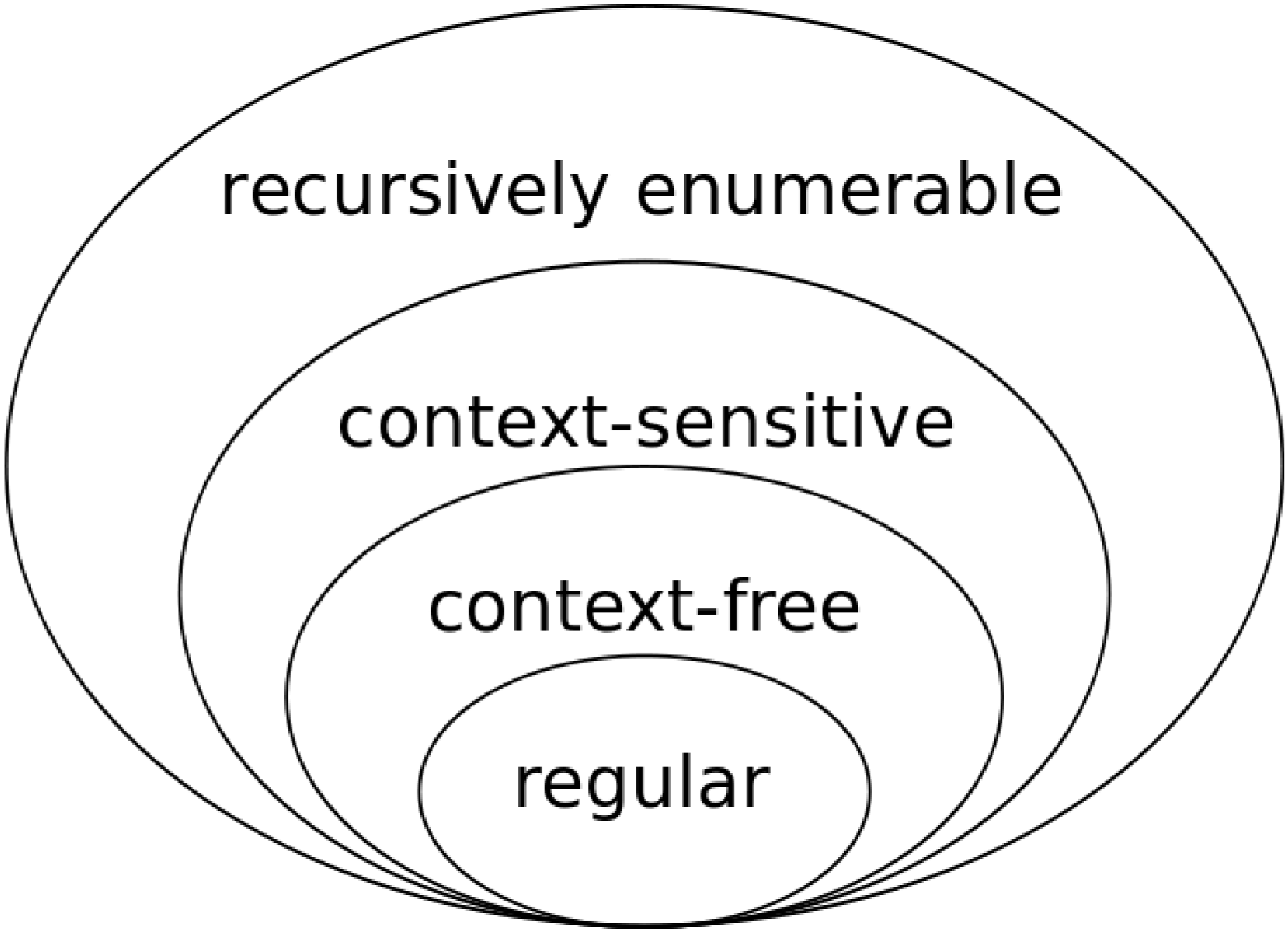
$$E \rightarrow (\ E\ ) \mid ()$$

Notes on Context-Free Languages

- It is undecidable if the language described by a context-free grammar is regular (Greibach's theorem);
- There exist languages that cannot be expressed by context-free grammars:

  $\{\mathtt{a}^{n}\mathtt{b}^{n}\mathtt{c}^{n} \mid n \geq 1\}$
- In parser construction we use a proper subset of context-free languages, namely *deterministic* context-free languages; and
- Such languages can be described by a *deterministic* pushdown automaton (same idea as DFA vs NFA, only one transition possible from a given state for an input/stack pair).
  - DPDAs cannot recognize all context-free languages!
  - **Example:** Even length palindrome $E \rightarrow$ a E a | b E b | ϵ. How do we know that matching should start?

Chomsky Hierarchy



https://en.wikipedia.org/wiki/Chomsky_hierarchy#/media/File:Chomsky-hierarchy.svg

Derivations

Given a context-free grammar, we can *derive* strings by repeatedly replacing variables with the RHS of a rule until only terminals remain (i.e. for a rewrite rule A $\rightarrow \gamma$, we replace A by γ). We begin with the start symbol.

Example

Derive the string "abc" using the following grammar and start symbol $\boldsymbol{A}$

$\boldsymbol{A} \rightarrow \boldsymbol{A}\ \boldsymbol{A} \mid \boldsymbol{B} \mid$ a

$\boldsymbol{B} \rightarrow$ b $\boldsymbol{B} \mid$ c

$\underline{\boldsymbol{A}}$

$\boldsymbol{A}\ \underline{\boldsymbol{A}}$

$\underline{\boldsymbol{A}}\ \boldsymbol{B}$

a $\underline{\boldsymbol{B}}$

a b $\underline{\boldsymbol{B}}$

a b c

A string is in the CFL if there exists a derivation using the CFG.

Derivations

Rightmost derivations and *leftmost derivations* expand the rightmost and leftmost non-terminals respectively until only terminals remain.

Example

Derive the string "abc" using the following grammar and start symbol $\boldsymbol{A}$

$\boldsymbol{A} \rightarrow \boldsymbol{A}\ \boldsymbol{A}$ | $\boldsymbol{B}$ | a

$\boldsymbol{B} \rightarrow$ b $\boldsymbol{B}$ | c

Rightmost	**Leftmost**
$\underline{\boldsymbol{A}}$	$\underline{\boldsymbol{A}}$
$\boldsymbol{A}\ \underline{\boldsymbol{A}}$	$\underline{\boldsymbol{A}}\ \boldsymbol{A}$
$\boldsymbol{A}\ \underline{\boldsymbol{B}}$	a $\underline{\boldsymbol{A}}$
$\boldsymbol{A}$ b $\underline{\boldsymbol{B}}$	a $\underline{\boldsymbol{B}}$
$\underline{\boldsymbol{A}}$ b c	a b $\underline{\boldsymbol{B}}$
a b c	a b c

Example Programming Language

CFG rules

Prog → Dcls Stmts

Dcls → Dcl Dcls | ϵ

Dcl → "int" ident | "float" ident

Stmts → Stmt Stmts | ϵ

Stmt → ident "=" Val

Val → num | ident

Corresponding Program

```
int a
float b
b = a
```

Leftmost derivation

$\underline{Prog}$

$\underline{Dcls}\ Stmts$

$\underline{Dcl}\ Dcls\ Stmts$

"int" ident $\underline{Dcls}\ Stmts$

"int" ident $\underline{Dcl}\ Dcls\ Stmts$

"int" ident "float" ident $\underline{Dcls}\ Stmts$

"int" ident "float" ident $\underline{Stmts}$

"int" ident "float" ident $\underline{Stmt}\ Stmts$

"int" ident "float" ident ident "=" $\underline{Val}\ Stmts$

"int" ident "float" ident ident "=" ident $\underline{Stmts}$

"int" ident "float" ident ident "=" ident

Backus-Naur Form (BNF)

```
stmt ::= stmt_expr ";" |
         while_stmt |
         block |
         if_stmt
while_stmt ::= WHILE "(" expr ")" stmt
block ::= "{" stmt_list "}"
if_stmt ::= IF "(" expr ")" stmt |
            IF "(" expr ")" stmt ELSE stmt
```

We have four options for `stmt_list`:

1. `stmt_list ::= stmt_list stmt |` ϵ (0 or more, left-recursive)
2. `stmt_list ::= stmt stmt_list |` ϵ (0 or more, right-recursive)
3. `stmt_list ::= stmt_list stmt | stmt` (1 or more, left-recursive)
4. `stmt_list ::= stmt stmt_list | stmt` (1 or more, right-recursive)

Extended BNF (EBNF)

Extended BNF provides '{' and '}' which act like Kleene *'s in regular expressions. Compare the following language definitions in BNF and EBNF

BNF	derivations		EBNF
$A \rightarrow A$ a | b	b	$\underline{A}$ a	$A \rightarrow$ b { a }
(left-recursive)		$\underline{A}$ a a	
		b a a	
$A \rightarrow$ a A | b	b	a $\underline{A}$	$A \rightarrow$ { a } b
(right-recursive)		a a $\underline{A}$	
		a a b	

EBNF Statement Lists

Using EBNF repetition, our four choices for `stmt_list`

1. `stmt_list ::= stmt_list stmt |` ϵ (0 or more, left-recursive)
2. `stmt_list ::= stmt stmt_list |` ϵ (0 or more, right-recursive)
3. `stmt_list ::= stmt_list stmt | stmt` (1 or more, left-recursive)
4. `stmt_list ::= stmt stmt_list | stmt` (1 or more, right-recursive)

can be reduced substantially since EBNF's {} does not specify a derivation order

1. `stmt_list ::= { stmt }`
2. `stmt_list ::= { stmt }`
3. `stmt_list ::= { stmt } stmt`
4. `stmt_list ::= stmt { stmt }`

ENBF Optional Construct

EBNF provides an optional construct using '`[`' and '`]`' which act like '?' in regular expressions.

A non-empty statement list (at least one element) in BNF

```
stmt_list ::= stmt stmt_list | stmt
```

can be re-written using the optional brackets as

```
stmt_list ::= stmt [ stmt_list ]
```

Similarly, an optional `else` block

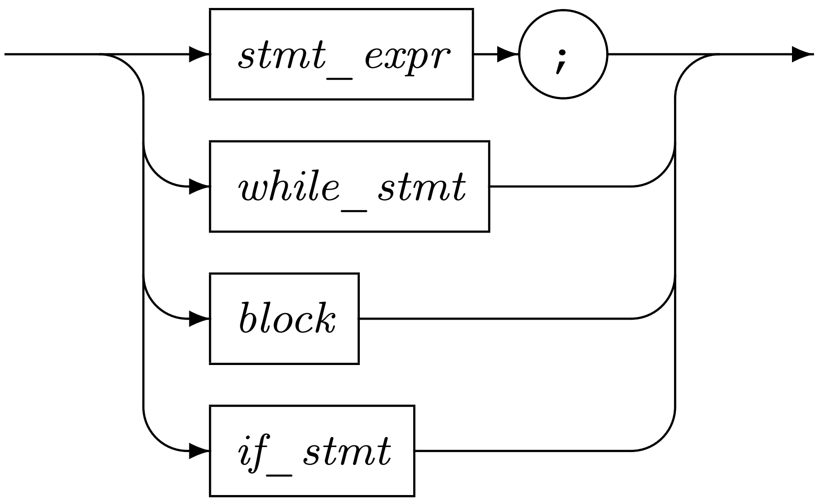
```
if_stmt ::= IF "(" expr ")" stmt |
            IF "(" expr ")" stmt ELSE stmt
```

can be simplified and re-written as

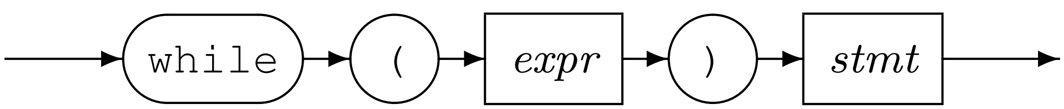
```
if_stmt ::= IF "(" expr ")" stmt [ ELSE stmt ]
```

Railroad Diagrams (thanks rail.sty!)

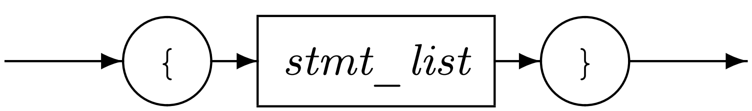
stmt



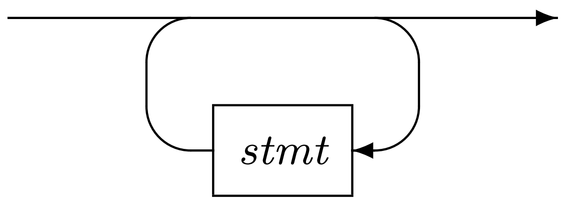
while_ stmt



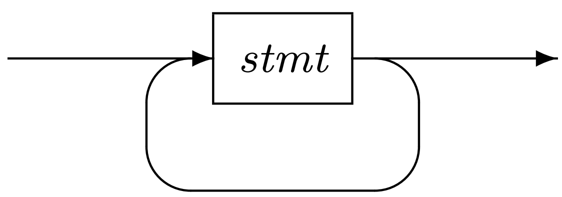
block

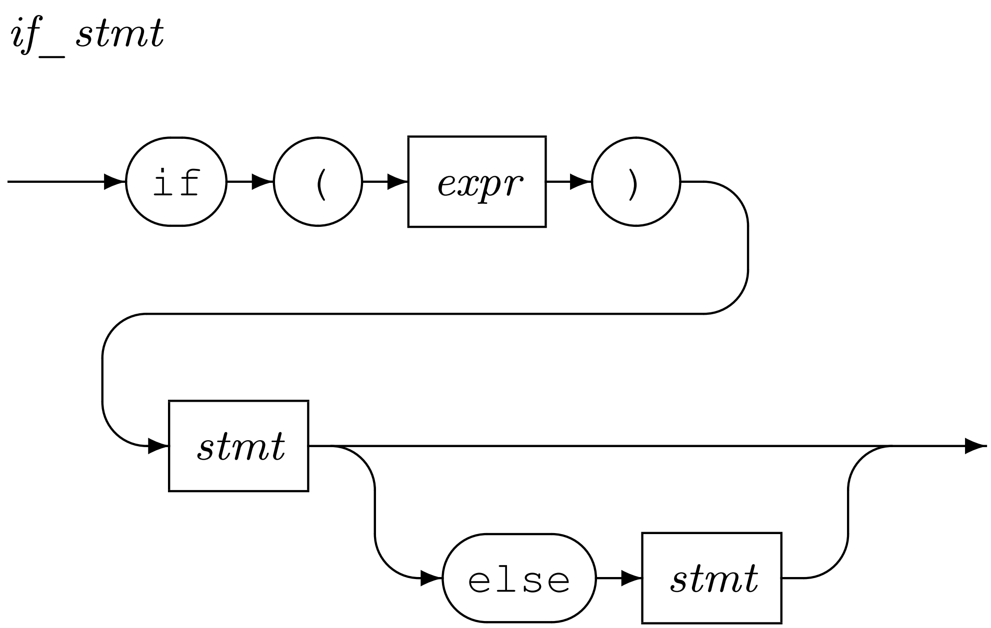


stmt_list (0 or more)



stmt_list (1 or more)





if_stmt
if
(
expr
)
stmt
else
stmt

Announcements (Wednesday, January 16th)

Milestones

- Continue picking your group (3 recommended). Who doesn't have a group?
- Learn `flex/bison` or `SableCC`

Assignment 1

- Reference compiler has been posted
- Any questions?
- **Due**: Friday, January 25th 11:59 PM

Midterm

- **Date**: February 26th from 6:00 - 7:30 PM in McConnell 103/321

Reference compiler (MiniLang)

Accessing

- `ssh <socs_username>@teaching.cs.mcgill.ca`
- `~cs520/minic {keyword} < {file}`
- If you find errors in the reference compiler, up to 5 bonus points on the assignment

Keywords for the first assignment

- `scan`: run scanner only, OK/Error
- `tokens`: produce the list of tokens for the program
- `parse`: run scanner+parser, OK/Error

Parse Tree

Given an input program P, the execution of a parser generates a *parse tree* (also called a *concrete syntax tree*) that

- Represents the syntax structure of a string; and
- Is built *exactly* from the rules given the context-free grammar.

Nodes in the tree

- Internal (parent) nodes represent the LHS of a rewrite rule;
- Child nodes represent the RHS of a rewrite rule.

The *fringe* (or leaves) or the tree form the sentence you derived.

Relationship with derivations

As the sentence is derived, the tree is formed

- Both rightmost and leftmost derivations give the same set of possible parse trees; but
- The order of forming nodes in the tree differs.

Example

Grammar

$S \rightarrow S\ ;\ S$

$S \rightarrow$ id := E

$S \rightarrow$ print (L)

$E \rightarrow$ id

$E \rightarrow$ num

$E \rightarrow E + E$

$E \rightarrow (S , E)$

$L \rightarrow E$

$L \rightarrow L , E$

Derive the following program using the above grammar

```
a := 7;
b := c + (d := 5 + 6, d)
```

Rightmost derivation

$\underline{S}$

S; $\underline{S}$

S; id := $\underline{E}$

S; id := E + $\underline{E}$

S; id := E + (S, $\underline{E}$)

S; id := E + ($\underline{S}$, id)

S; id := E + (id := $\underline{E}$, id)

S; id := E + (id := E + $\underline{E}$, id)

S; id := E + (id := $\underline{E}$ + num, id)

S; id := $\underline{E}$ + (id := num + num, id)

$\underline{S}$; id := id + (id := num + num, id)

id := $\underline{E}$; id := id + (id := num + num, id)

id := num; id := id + (id := num + num, id)

Example

Rightmost derivation

$\underline{\boldsymbol{S}}$

$\boldsymbol{S}$; $\underline{\boldsymbol{S}}$

$\boldsymbol{S}$; id := $\underline{\boldsymbol{E}}$

$\boldsymbol{S}$; id := $\boldsymbol{E}$ + $\underline{\boldsymbol{E}}$

$\boldsymbol{S}$; id := $\boldsymbol{E}$ + ($\boldsymbol{S}$, $\underline{\boldsymbol{E}}$)

$\boldsymbol{S}$; id := $\boldsymbol{E}$ + ($\underline{\boldsymbol{S}}$, id)

$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id := $\underline{\boldsymbol{E}}$, id)

$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id := $\boldsymbol{E}$ + $\underline{\boldsymbol{E}}$, id)

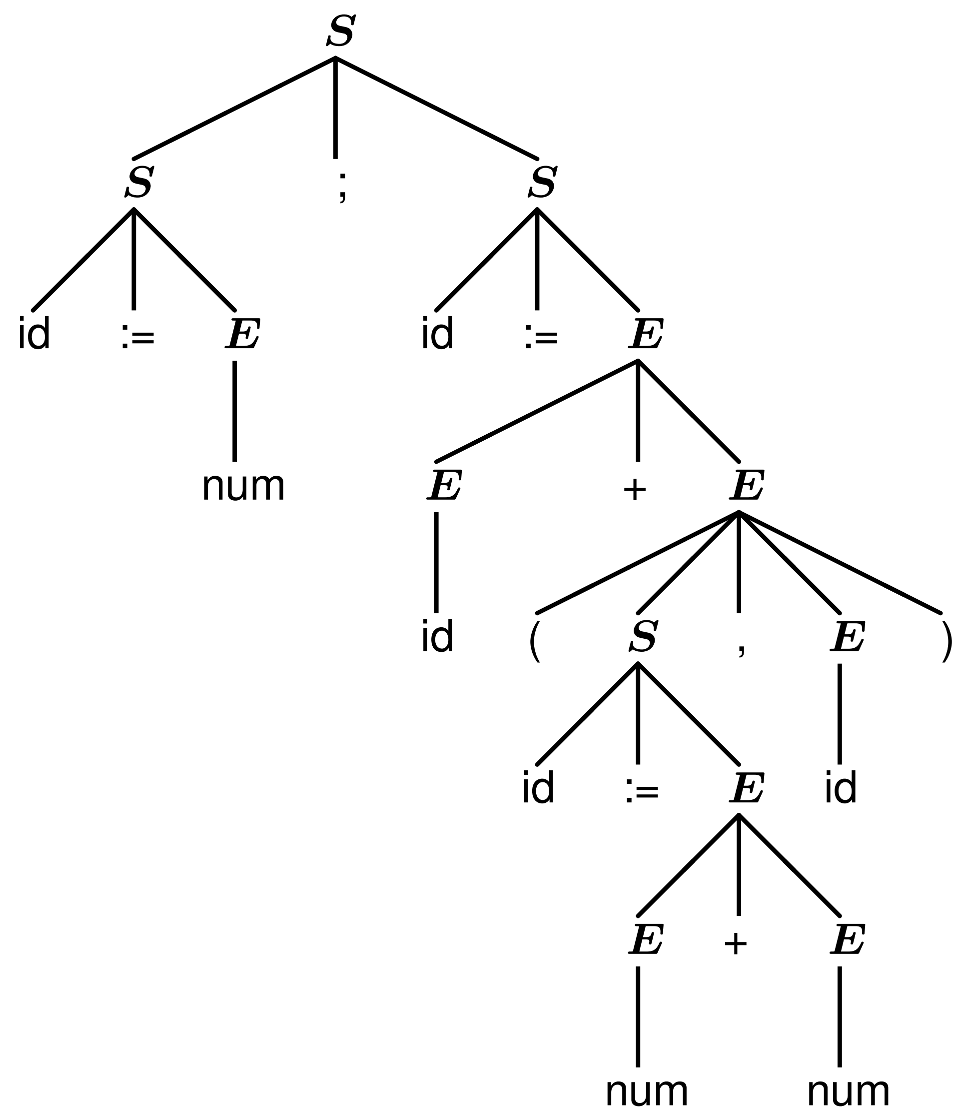
$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id := $\underline{\boldsymbol{E}}$ + num, id)

$\boldsymbol{S}$; id := $\underline{\boldsymbol{E}}$ + (id := num + num, id)

$\underline{\boldsymbol{S}}$; id := id + (id := num + num, id)

id := $\underline{\boldsymbol{E}}$; id := id + (id := num + num, id)

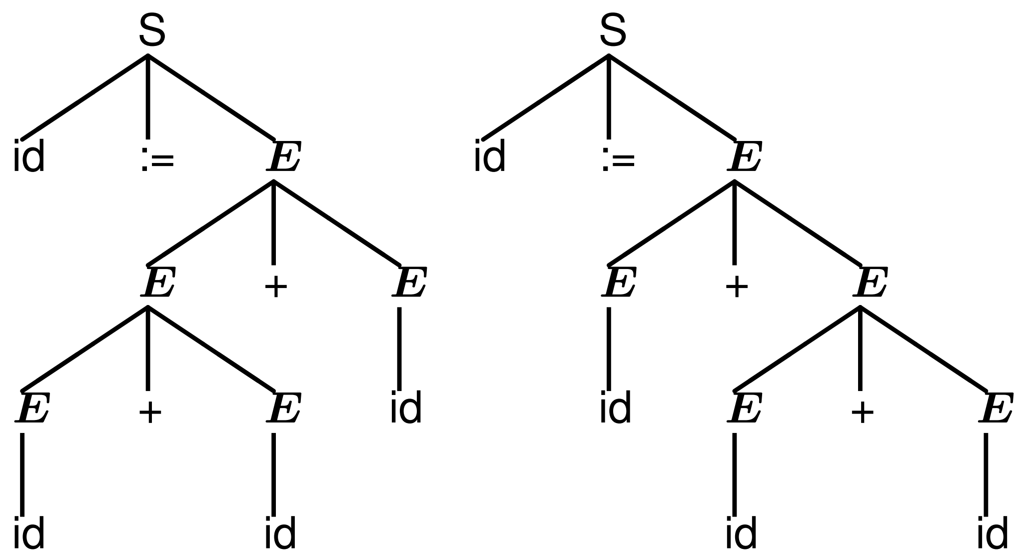
id := num; id := id + (id := num + num, id)



Ambiguous Grammars

A grammar is *ambiguous* if a sentence more than one parse tree

```
id := id + id + id
```



The above is harmless, but consider operations whose order matters

```
id := id - id - id
id := id + id * id
```

Clearly, we need to consider associativity and precedence when designing grammars.

Ambiguous Grammars

Ambiguous grammars can have severe consequences parsing for programming languages

- Not all context-free languages have an unambiguous grammar (COMP 330);
- Deterministic pushdown automata that are used by parsers *require* an unambiguous grammar.

We must therefore carefully design our languages and grammar to avoid ambiguity.

How can we make grammars unambiguous?

Assuming our language has rules to handle ambiguities we can

- Manually rewrite the grammar to be unambiguous; or
- Use precedence rules to resolve ambiguities.

For this class you should understand how to identify and resolve ambiguities using both approaches.

Rewriting an Ambiguous Grammar

Given the following expression grammar, what ambiguities exist?

$E \rightarrow E + E$	$E \rightarrow E * E$	$E \rightarrow$ id
$E \rightarrow E - E$	$E \rightarrow E \;/\; E$	$E \rightarrow$ num
		$E \rightarrow (E)$

Ambiguities

Ambiguities exist when there is more than one way of parsing a given expression (there exists more than one unique parse tree)

- Grouping of operands between operations of different precedence (BEDMAS); or
- Grouping of operands between operations of the same precedence.

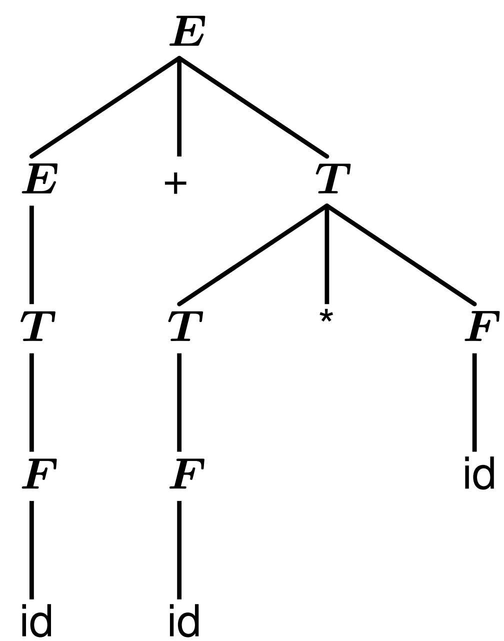
Rewriting an Ambiguous Grammar

Given an ambiguous grammar for expressions (refer to the previous slides for details)

$$E \rightarrow E + E \qquad E \rightarrow E * E \qquad E \rightarrow \text{id}$$
$$E \rightarrow E - E \qquad E \rightarrow E \;/\; E \qquad E \rightarrow \text{num}$$
$$E \rightarrow (\; E \;)$$

We can rewrite (factor) the grammar using *terms* and *factors* to become unambiguous

$$E \rightarrow E + T \qquad T \rightarrow T * F \qquad F \rightarrow \text{id}$$
$$E \rightarrow E - T \qquad T \rightarrow T \;/\; F \qquad F \rightarrow \text{num}$$
$$E \rightarrow T \qquad T \rightarrow F \qquad F \rightarrow (\; E \;)$$



Why does this work?

Rewriting an Ambiguous Grammar

Expression grammars must have 2 mathematical attributes for operations

- **Precedence**: Order of operations ($\star$ and $/$ have precendence over $+$ and $-$)
- **Associativity**: Grouping of operations with the same precedence

Rewriting

These attributes are imposed through “constraints” that we build into the grammar

- Operands (LHS/RHS) of one operation must not expand to other operations of lower precedence;
- If an operation is left-associative, then *only* its LHS may expand to an operation of equal or higher precedence; and
- If an operation is right-associative, then *only* its RHS may expand to an operation of equal or higher precedence.

The Dangling Else Problem

The dangling else problem is another well known parsing challenge with nested if-statements. Given the grammar, where IfStmt is a valid statement

IfStmt → `tIF` Expr `tTHEN` Stmt `tELSE` Stmt
  | `tIF` Expr `tTHEN` Stmt

Consider the following program (left) and token stream (right)

```
if {expr} then
    if {expr} then
        <stmt>
else
    <stmt>
```

```
tIF
Expr
tTHEN
tIF
Expr
tTHEN
Stmt
tELSE
Stmt
```

To which if-statement does the else (and corresponding statement) belong?

The issue arises because the if-statement does not have a termination (endif), and braces are not required for the branches.

Parsers

- Take a string of tokens generated by the scanner as input; and
- Build a parse tree according to some grammar.
- *In a theoretical sense, parsing checks that a string is contained in a language*

Types of parsers

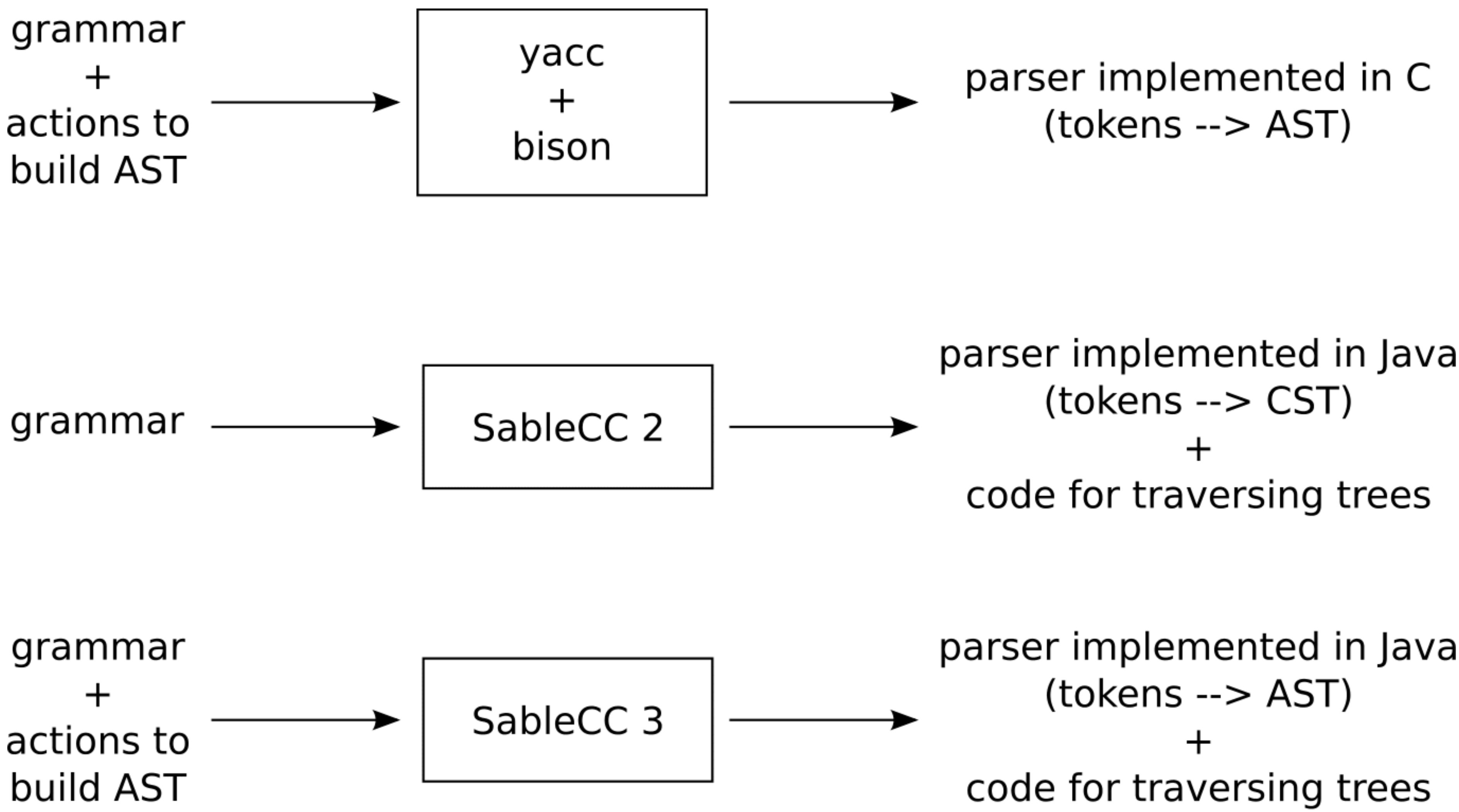
1. Top-down, *predictive* or *recursive descent* parsers. Used in all languages designed by Wirth, e.g. Pascal, Modula, and Oberon; and
2. Bottom-up parsers.

Automated Parser Generators

Writing the parser for a large context-free language is lengthy! Automated parser generators exist which

- Use (deterministic) context-free grammars as input; and
- Generate parsers using the machinery of a deterministic pushdown automaton.

(LALR) Parser Tools



`bison` (previously `yacc`)

`bison` is a parser generator that

- Takes a grammar as input;
- Computes an LALR(1) parser table;
- Reports conflicts (if any);
- Potentially resolves conflicts using defaults (!!); and
- Creates a parser written in C.

Warning!

Be sure to resolve conflicts, otherwise you may end up with difficult to find parsing errors

Example `bison` File

The expression grammar given below is expressed in `bison` as follows

$E \rightarrow \text{id}$ $\quad E \rightarrow E * E$ $\quad E \rightarrow E + E$ $\quad E \rightarrow (E)$

$E \rightarrow \text{num}$ $\quad E \rightarrow E \,/\, E$ $\quad E \rightarrow E - E$

```
%{ /* C declarations */ %}

/* Bison declarations; tokens come from lexer (scanner) */
%token tIDENTIFIER tINTVAL

/* Grammar rules after the first %% */
%start exp
%%
exp : tIDENTIFIER
    | tINTVAL
    | exp '*' exp
    | exp '/' exp
    | exp '+' exp
    | exp '-' exp
    | '(' exp ')'
;
%% /* User C code after the second %% */
```

`bison` Conflicts

As we previously discussed, the basic expression grammar is ambiguous.

`bison` reports cases where more than one parse tree is possible as `shift/reduce` or `reduce/reduce` conflicts – we will see more about this later!

```
$ bison --verbose tiny.y # --verbose produces tiny.output
tiny.y contains 16 shift/reduce conflicts.
```

Using the `--verbose` option we can output a full diagnostics log

```
$ cat tiny.output
State 11 contains 4 shift/reduce conflicts.
State 12 contains 4 shift/reduce conflicts.
State 13 contains 4 shift/reduce conflicts.
State 14 contains 4 shift/reduce conflicts.

[...]
```

`bison` Resolving Conflicts (Rewriting)

The first option in `bison` involves rewriting the grammar to resolve ambiguities (terms/factors)

$$\boldsymbol{E} \rightarrow \boldsymbol{E} + \boldsymbol{T} \qquad \boldsymbol{T} \rightarrow \boldsymbol{T} * \boldsymbol{F} \qquad \boldsymbol{F} \rightarrow \text{id}$$

$$\boldsymbol{E} \rightarrow \boldsymbol{E} - \boldsymbol{T} \qquad \boldsymbol{T} \rightarrow \boldsymbol{T} \,/\, \boldsymbol{F} \qquad \boldsymbol{F} \rightarrow \text{num}$$

$$\boldsymbol{E} \rightarrow \boldsymbol{T} \qquad \boldsymbol{T} \rightarrow \boldsymbol{F} \qquad \boldsymbol{F} \rightarrow (\boldsymbol{E})$$

```
%token tIDENTIFIER tINTVAL

%start exp
%%
exp : exp '+' term
    | exp '-' term
    | term
;
term : term '*' factor
     | term '/' factor
     | factor
;
factor : tIDENTIFIER
       | tINTVAL
       | '(' exp ')'
;
```

bison Resolving Conflicts (Directives)

bison also provides *precedence directives* which automatically resolve conflicts

```
%token tIDENTIFIER tINTVAL

%left '+' '-'    /* left-associative, lower precedence */
%left '*' '/'    /* left-associative, higher precedence */

%start exp
%%
exp : tIDENTIFIER
    | tINTVAL
    | exp '*' exp
    | exp '/' exp
    | exp '+' exp
    | exp '-' exp
    | '(' exp ')'
;
```

`bison` Resolving Conflicts (Directives)

The conflicts are automatically resolved using either *shifts* or *reduces* depending on the directive.

- `%left` *(left-associative)*
- `%right` *(right-associative)*
- `%nonassoc` *(non-associative)*

Precedences are ordered from lowest to highest on a linewise basis.

Note: Although we only cover their use for expression grammars, precedence directives can be used for other ambiguities

Example `bison` File

```
%{
        #include <stdio.h>
        void yyerror(const char *s) { fprintf(stderr, "Error: %s\n", s); }
%}
%error-verbose

%union {
        int intval;
        char *identifier;
}

%token <intval> tINTVAL
%token <identifier> tIDENTIFIER

%left '+' '-'
%left '*' '/'

%start exp
%%
exp : tIDENTIFIER { printf("Load %s\n", $1); }
    | tINTVAL     { printf("Push %i\n", $1); }
    | exp '*' exp { printf("Mult\n"); }
    | exp '/' exp { printf("Div\n"); }
    | exp '+' exp { printf("Plus\n"); }
    | exp '-' exp { printf("Minus\n"); }
    | '(' exp ')' {}
;
%%
```

Example `flex` File

```
%{
        #include "y.tab.h"  /* Token types */
        #include <stdlib.h> /* atoi */
%}
DIGIT [0-9]
%option yylineno

%%
[ \t\n\r]+
"*"       return '*';
"/"       return '/';
"+"       return '+';
"-"       return '-';
"("       return '(';
")"       return ')';
0|([1-9]{DIGIT}*) {
        yylval.intval = atoi(yytext);
        return tINTVAL;
}
[a-zA-Z_][a-zA-Z0-9_]* {
        yylval.identifier = strdup(yytext);
        return tIDENTIFIER;
}

. { fprintf(stderr, "Error: (line %d) unexpected char '%s'\n", yylineno, yytext);
    exit(1);
}
%%
```

Running a `bison+flex` Scanner and Parser

After the scanner file is complete, using `flex`/`bison` to create the parser is really simple

```
$ flex tiny.l # generates lex.yy.c
$ bison --yacc tiny.y # generates y.tab.h/c
$ gcc lex.yy.c y.tab.c y.tab.h main.c -o tiny -lfl
```

Note that we provide a main file which calls the parser (`yyparse()`)

```
void yyparse();
int main(void)
{
        yyparse();
        return 0;
}
```

Example

Running the example scanner on input `a*(b-17) + 5/c` yields

```
$ echo "a*(b-17) + 5/c" | ./tiny
```

```
Load a
Load b
Push 17
Minus
Mult
Push 5
Load c
Div
Plus
```

Which is the correct order of operations. You should confirm this for yourself!

Error Recovery

If the input contains syntax errors, then the `bison`-generated parser calls `yyerror` and stops.

We may ask it to recover from the error by having a production with `error`

```
exp : tIDENTIFIER { printf ("Load %s\n", $1); }
...
    | '(' exp ')'
    | error { yyerror(); }
;
```

and on input `a@(b-17) ++ 5/c` we get the output

```
Load a
Syntax error before (
Syntax error before (
Syntax error before (
Syntax error before b
Push 17
Minus
Syntax error before )
Syntax error before )
Syntax error before +
Plus
Push 5
Load c
Div
Plus
```

Unary Minus

A unmary minus has highest precedence - we expect the expression `-5 * 3` to be parsed as `(-5) * 3` rather than `-(5 * 3)`

To encourage `bison` to behave as expected, we use precedence directives with a special unused token

GRAMMAR 3.37: Yacc grammar with precedence directives.

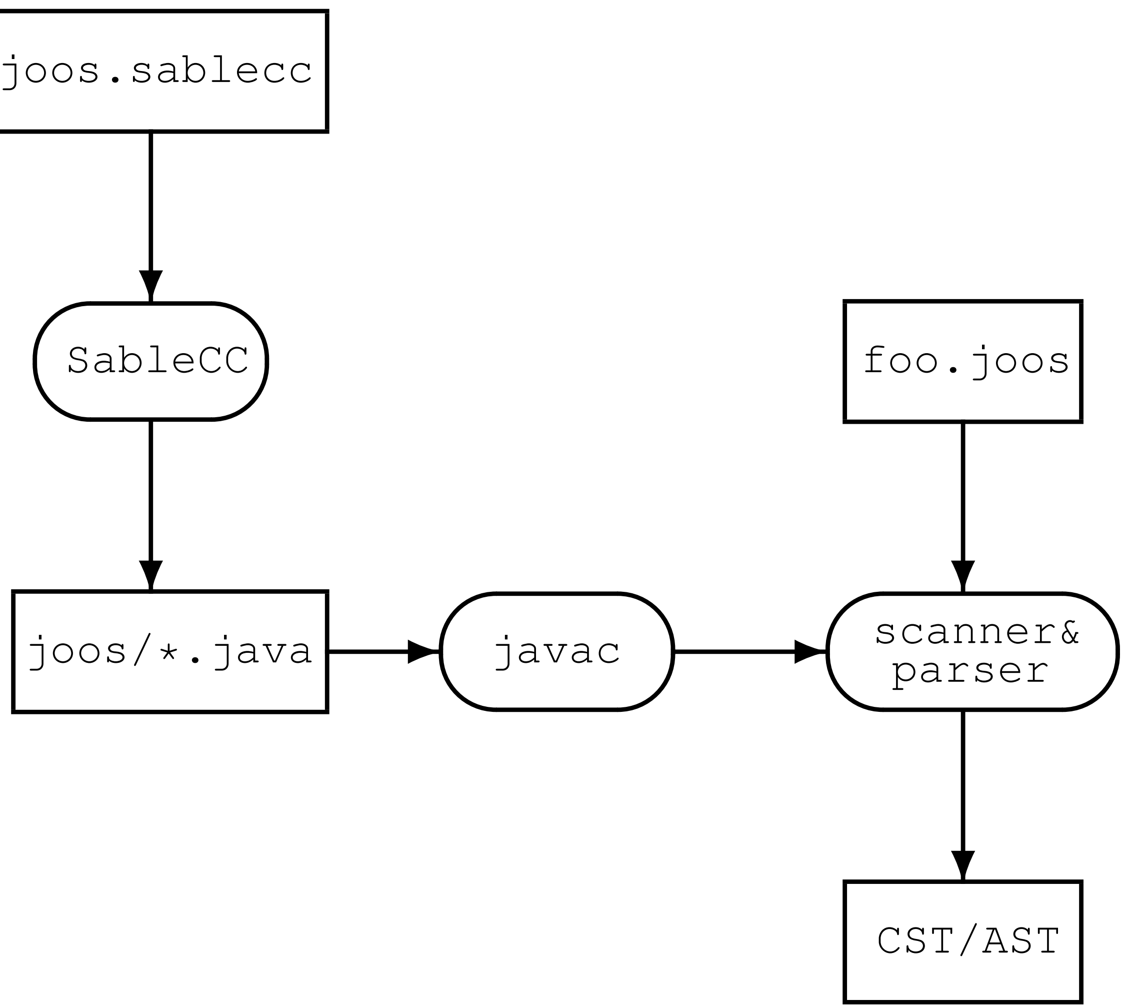
```
%{ declarations of yylex and yyerror %}
%token INT PLUS MINUS TIMES UMINUS
%start exp

%left PLUS MINUS
%left TIMES
%left UMINUS
%%

exp : INT
    | exp PLUS exp
    | exp MINUS exp
    | exp TIMES exp
    | MINUS exp    %prec UMINUS
```

SableCC

SableCC (by Etienne Gagnon, McGill alumnus) is a *compiler compiler*: it takes a grammatical description of the source language as input, and generates a lexer (scanner) and parser.



SableCC 2 Example

Scanner definition

```
Package tiny;
Helpers
  tab   = 9;
  cr    = 13;
  lf    = 10;
  digit = ['0'..'9'];
  lowercase = ['a'..'z'];
  uppercase = ['A'..'Z'];
  letter  = lowercase | uppercase;
  idletter = letter | '_';
  idchar  = letter | '_' | digit;
Tokens
  eol   = cr | lf | cr lf;
  blank = ' ' | tab;
  star  = '*';
  slash = '/';
  plus  = '+';
  minus = '-';
  l_par = '(';
  r_par = ')';
  number  = '0'| [digit-'0'] digit*;
  id    = idletter idchar*;
Ignored Tokens
  blank, eol;
```

SableCC 2 Example

Parser definition

```
Productions
  exp =
      {plus}    exp plus factor |
      {minus}   exp minus factor |
      {factor}  factor;

  factor =
      {mult}    factor star term |
      {divd}    factor slash term |
      {term}    term;

  term =
      {paren}   l_par exp r_par |
      {id}      id |
      {number}  number;
```

Sable CC version 2 produces parse trees, a.k.a. concrete syntax trees (CSTs).

SableCC 3 Grammar

```
Productions
  cst_exp {-> exp} =
      {cst_plus}    cst_exp plus factor
                    {-> New exp.plus(cst_exp.exp,factor.exp)} |
      {cst_minus}   cst_exp minus factor
                    {-> New exp.minus(cst_exp.exp,factor.exp)} |
      {factor}      factor {-> factor.exp};

  factor {-> exp} =
      {cst_mult}    factor star term
                    {-> New exp.mult(factor.exp,term.exp)} |
      {cst_divd}    factor slash term
                    {-> New exp.divd(factor.exp,term.exp)} |
      {term}        term {-> term.exp};

  term {-> exp} =
      {paren}       l_par cst_exp r_par {-> cst_exp.exp} |
      {cst_id}      id {-> New exp.id(id)} |
      {cst_number}  number {-> New exp.number(number)};
```

SableCC version 3 allows the compiler writer to generate abstract syntax trees (ASTs).

SableCC 3 AST Definition

```
Abstract Syntax Tree
  exp =
      {plus}      [l]:exp [r]:exp |
      {minus}     [l]:exp [r]:exp |
      {mult}      [l]:exp [r]:exp |
      {divd}      [l]:exp [r]:exp |
      {id}        id |
      {number}    number;
```

Announcements (Friday, January 18th)

Milestones

- Continue picking your group (3 recommended). Who doesn't have a group?
- Learn `flex`/`bison` or `SableCC`

Assignment 1

- Any questions?
- **Due**: Friday, January 25th 11:59 PM

Midterm

- **Date**: February 26th from 6:00 - 7:30 PM in McConnell 103/321

Reference compiler (MiniLang)

Accessing

- `ssh <socs_username>@teaching.cs.mcgill.ca`
- `~cs520/minic {keyword} < {file}`
- If you find errors in the reference compiler, up to 5 bonus points on the assignment

Keywords for the first assignment

- `scan`: run scanner only, OK/Error
- `tokens`: produce the list of tokens for the program
- `parse`: run scanner+parser, OK/Error

Top-Down Parsers

- Can (easily) be written by hand; or
- Generated from an LL(*k*) grammar:
  - *Left-to-right parse*;
  - *Leftmost-derivation*; and
  - *k symbol lookahead*.
- **Algorithm idea:** an LL(k) parser takes the leftmost non-terminal A, looks at *k* tokens of lookahead, and determines which rule $A \rightarrow \gamma$ should be used to replace A
  - Begin with the start symbol (root);
  - Grows the parse tree using the defined grammar; by
  - *Predicting*: the parser must determine (given some input) which rule to apply next.

Example of LL(1) Parsing

Grammar

Prog $\rightarrow$ Dcls Stmts

Dcls $\rightarrow$ Dcl Dcls | ϵ

Dcl $\rightarrow$ `"int"` ident | `"float"` ident

Stmts $\rightarrow$ Stmt Stmts | ϵ

Stmt $\rightarrow$ ident `"="` Val

Val $\rightarrow$ num | ident

Scanner token string

```
tINT
tIDENTIFIER(a)
tFLOAT
tIDENTIFIER(b)
tIDENTIFIER(b)
tASSIGN
tIDENTIFIER(a)
```

Parse the program

```
int a
float b
b = a
```

Example of LL(1) Parsing

Derivation	Next Token	Options
Prog	`tINT`	Dcls Stmts
Dcls Stmts	`tINT`	Dcl Dcls | ϵ
Dcl Dcls Stmts	`tINT`	"int" ident | "float" ident
"int" ident Dcls Stmts	`tFLOAT`	Dcl Dcls | ϵ
"int" ident Dcl Dcls Stmts	`tFLOAT`	"int" ident | "float" ident
"int" ident "float" ident Dcls Stmts	`tIDENTIFIER`	Dcl Dcls | ϵ
"int" ident "float" ident Stmts	`tIDENTIFIER`	Stmt Stmts | ϵ
"int" ident "float" ident Stmt Stmts	`tIDENTIFIER`	ident "=" Val
"int" ident "float" ident ident "=" Val Stmts	`tIDENTIFIER`	num | ident
"int" ident "float" ident ident "=" ident Stmts	`EOF`	Stmt Stmts | ϵ
"int" ident "float" ident ident "=" ident		

Notes on LL(1) Parsing

In the previous example, each step of the parser

- Determined the next rule looking at exactly 1 token of the input stream; and
- Only has one possible rule to apply given the token.

The grammar is therefore LL(1) and can be used by LL(1) parsing tools.

Limitations

However, not all grammars are LL(1). In fact, not all grammars are LL(k) for *any* fixed k

- LL(k) grammars have a fixed lookahead; but
- Deciding between some rules might require *unbounded* lookahead.

Recursive Descent Parsers

Recursive descent parsers use a set of mutually recursive functions (1 per non-terminal) for parsing

Idea: Repeatedly expand the leftmost non-terminal by *predicting* which rule to use.

- Each rule for a non-terminal has a *predict set* that indicates if the rule can be applied given the *k* lookahead tokens; and
- If the next tokens are in
  - Exactly one of the predict sets: the corresponding rule is applied;
  - More than one of the predict sets: there is a conflict; or
  - None of the predict sets: there is a syntax error.
- Applying the rules/productions
  - Consume/match terminals; and
  - Recursively call functions for other non-terminals.

Recursive Descent Example

Given a subset of the previous context-free grammar

Prog $\longrightarrow$ Dcls Stmts

Dcls $\longrightarrow$ Dcl Dcls | ϵ

Dcl $\longrightarrow$ `"int"` ident | `"float"` ident

We can define predict sets for all rules, giving us the following recursive descent parser functions

```
function Prog()
    call Dcls()
    call Stmts()
end

function Dcls()
    switch nextToken()
        case tINT|tFLOAT:
            call Dcl()
            call Dcls()
        case tIDENT|EOF:
            /* no more declarations, parsing
            continues in the Prog method */
            return
    end
end
```

```
function Dcl()
    switch nextToken()
        case tINT:
            match(tINT)
            match(tIDENTIFIER)
        case tFLOAT:
            match(tFLOAT)
            match(tIDENTIFIER)
    end
end
```

Common Prefixes

While this approach to parsing is simple and intuitive, it has its limitations. Consider the following productions, defining an If-Else-End construct

IfStmt $\rightarrow$ `tIF` Exp `tTHEN` Stmts `tEND`

| `tIF` Exp `tTHEN` Stmts `tELSE` Stmts `tEND`

With bounded lookahead (say an LL(1) parser), we are unable to predict which rule to follow as both rules have {`tIF`} as their predict set.

Solution

To resolve this issue, we *factor* the grammar

IfStmt $\rightarrow$ `tIF` Exp `tTHEN` Stmts IfEnd

IfEnd $\rightarrow$ `tEND` | `tELSE` Stmts `tEND`

There is now only a single IfStmt rule and thus no ambiguity. Additionally, productions for the IfEnd variable have non-intersecting predict sets

1. {`tEND`}
2. {`tELSE`}

Recursive Lists

In context-free grammars, we define lists recursively. The following rules specify lists of 0 or more and 1 or more elements respectively

$$A \rightarrow A\ \beta \mid \epsilon$$

$$B \rightarrow B\ \beta \mid \beta$$

$$\beta \rightarrow \texttt{tTOKEN}$$

They are also *left-recursive*, as the recursion occurs on the left hand side. We can similarly define right-recursive grammars by swapping the order of the elements

$$A \rightarrow \beta\ A \mid \epsilon$$

$$B \rightarrow \beta\ B \mid \beta$$

Using the above grammars, deriving the sentence `tTOKEN` is simple.

Left Recursion

Left recursion also causes difficulties with LL(*k*) parsers. Consider the following productions

$\boldsymbol{A} \rightarrow \boldsymbol{A}\ \boldsymbol{\beta} \mid \epsilon$

$\boldsymbol{\beta} \rightarrow$ `tTOKEN`

Assume we can come up with a predict set for $\boldsymbol{A}$ consisting of `tTOKEN`, then applying this rule gives

Expansion	Next Token
$\underline{\boldsymbol{A}}$	`tTOKEN`
$\underline{\boldsymbol{A}}\ \boldsymbol{\beta}$	`tTOKEN`
$\underline{\boldsymbol{A}}\ \boldsymbol{\beta}\ \boldsymbol{\beta}$	`tTOKEN`
$\underline{\boldsymbol{A}}\ \boldsymbol{\beta}\ \boldsymbol{\beta}\ \boldsymbol{\beta}$	`tTOKEN`
$\underline{\boldsymbol{A}}\ \boldsymbol{\beta}\ \boldsymbol{\beta}\ \boldsymbol{\beta}\ \boldsymbol{\beta}$	`tTOKEN`
$\underline{\boldsymbol{A}}\ \boldsymbol{\beta}\ \boldsymbol{\beta}\ \boldsymbol{\beta}\ \boldsymbol{\beta}\ \boldsymbol{\beta}$	`tTOKEN`
...	

This continues on forever. *Note there are other ways to think of this as shown in the textbook*

The Dangling Else Problem - LL

To resolve this ambiguity we wish to associate the else with the *nearest unmatched* if-statement.

```
if {expr} then
    if {expr} then
        <stmt>
else
    <stmt>
```

```
[if {expr} then
    [if {expr} then
        <stmt>
else
    <stmt>]]
```

Note that any grammar we come up with is still not LL(*k*). Why not?

Recursive Descent Parsing

Even though we cannot write an LL(*k*) grammar, it is easy to write a recursive descent parser using a greedy-ish approach to matching.

```
function Stmt()
    switch nextToken():
        case tIF:
            call IfStmt()
        [...]
end
```

```
function IfStmt()
    match(tIF)
    call Expr()
    match(tTHEN)
    call Stmt()
    if nextToken() == tELSE:
        match(tELSE)
        call Stmt()
end
```

Bottom-Up Parsers

- Can be written by hand (tricky); or
- Generated from an LR(k) grammar (easy):
  - *Left-to-right parse*;
  - *Rightmost-derivation*; and
  - *k symbol lookahead*.
- **Algorithm idea**: form the parse tree by repeatedly grouping terminals and non-terminals into non-terminals until they form the root (start symbol).

Bottom-Up Parsers

- Build parse trees from the leaves to the root;
- Perform a rightmost derivation in reverse; and
- Use productions to replace the RHS of a rule with the LHS.

This is the *opposite* of a top-down parser.

The techniques used by bottom-up parsers are more complex to understand, but can use a larger set of grammars to top-down parsers.

Shift-Reduce Bottom-Up Parsing

Grammar

A shift-reduce parser starts with an extended grammar

- Introduce a new start symbol S' and an end-of-file token \$; and
- Form a new rule $S' \rightarrow S$ \$.

Practically, this ensures that the parser knows the end of input and no tokens may be ignored.

$S' \rightarrow S\$$	$S \rightarrow S$; S	$E \rightarrow$ id	$L \rightarrow E$
	$S \rightarrow$ id := E	$E \rightarrow$ num	$L \rightarrow L$, E
	$S \rightarrow$ print (L)	$E \rightarrow E$ + E	
		$E \rightarrow$ (S , E)	

Shift-Reduce Bottom-Up Parsing

Stack and Input

A shift-reduce parser maintains 2 collections of tokens

1. The input stream from the scanner
2. A work-in-progress stack represents subtrees formed over the currently parsed elements (terminals and non-terminals)

Actions

We then define the following actions

- **Shift**: move the first token from the input stream to top of the stack
- **Reduce:** replace α (a sequence of terminals/non-terminals) on the top of stack by X using rule $X \rightarrow \alpha$
- **Accept:** when S' is on the stack

Shift-Reduce Example

	`a:=7; b:=c+(d:=5+6,d)$`	shift
id	`:=7; b:=c+(d:=5+6,d)$`	shift
id :=	`7; b:=c+(d:=5+6,d)$`	shift
id := num	`; b:=c+(d:=5+6,d)$`	$E \rightarrow$ num
id := E	`; b:=c+(d:=5+6,d)$`	$S \rightarrow$ id:=E
S	`; b:=c+(d:=5+6,d)$`	shift
S;	`b:=c+(d:=5+6,d)$`	shift
S; id	`:=c+(d:=5+6,d)$`	shift
S; id :=	`c+(d:=5+6,d)$`	shift
S; id := id	`+(d:=5+6,d)$`	$E \rightarrow$ id
S; id := E	`+(d:=5+6,d)$`	shift
S; id := E +	`(d:=5+6,d)$`	shift
S; id := E + (	`d:=5+6,d)$`	shift
S; id := E + (id	`:=5+6,d)$`	shift
S; id := E + (id :=	`5+6,d)$`	shift
S; id := E + (id := num	`+6,d)$`	$E \rightarrow$ num
S; id := E + (id := E	`+6,d)$`	shift
S; id := E + (id := E +	`6,d)$`	shift
S; id := E + (id := E + num	`,d)$`	$E \rightarrow$ num
S; id := E + (id := E + E	`,d)$`	$E \rightarrow E$+E

Shift-Reduce Example (Continued)

S; id := E + (id := E + E	, d) $	$E \rightarrow E$+E
S; id := E + (id := E	, d) $	$S \rightarrow$ id:=E
S; id := E + (S	, d) $	shift
S; id := E + (S,	d) $	shift
S; id := E + (S, id	) $	$E \rightarrow$ id
S; id := E + (S, E	) $	shift
S; id := E + (S, E)	$	$E \rightarrow$ (S;E)
S; id := E + E	$	$E \rightarrow E$+E
S; id := E	$	$S \rightarrow$ id:=E
S; S	$	$S \rightarrow S$;S
S	$	shift
$S$$		$S' \rightarrow S$$
S'		accept

Shift-Reduce Rules (Example)

Recall the previous rightmost derivation of the string

```
a := 7;
b := c + (d := 5 + 6, d)
```

Rightmost derivation:

$\underline{\boldsymbol{S}}$

$\boldsymbol{S}$; $\underline{\boldsymbol{S}}$

$\boldsymbol{S}$; id := $\underline{\boldsymbol{E}}$

$\boldsymbol{S}$; id := $\boldsymbol{E}$ + $\underline{\boldsymbol{E}}$

$\boldsymbol{S}$; id := $\boldsymbol{E}$ + ($\boldsymbol{S}$, $\underline{\boldsymbol{E}}$)

$\boldsymbol{S}$; id := $\boldsymbol{E}$ + ($\underline{\boldsymbol{S}}$, id)

$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id := $\underline{\boldsymbol{E}}$, id)

$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id := $\boldsymbol{E}$ + $\underline{\boldsymbol{E}}$, id)

$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id := $\underline{\boldsymbol{E}}$ + num, id)

$\boldsymbol{S}$; id := $\underline{\boldsymbol{E}}$ + (id := num + num, id)

$\underline{\boldsymbol{S}}$; id := id + (id := num + num, id)

id := $\underline{\boldsymbol{E}}$; id := id + (id := num + num, id)

id := num; id := id + (id := num + num, id)

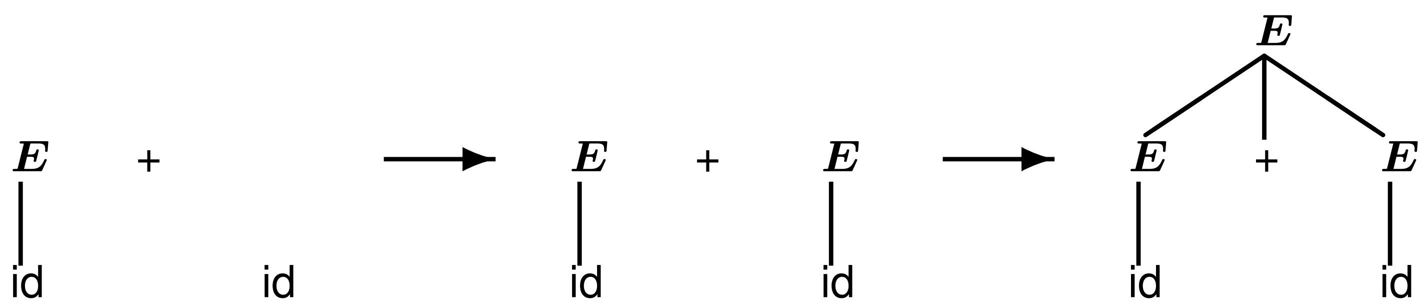
Note that the rules applied in LR parsing are the same as those above, *in reverse*.

Shift-Reduce Rules (Intuition)

To reduce using rule $A \rightarrow \gamma$, a shift-reduce parser must wait until the *top* of the stack contains *all* elements of γ.

If we think about shift-reduce in terms of parse trees

- Stack contains multiple subtrees (i.e. a forest); and
- Reduce actions take subtrees in γ and form new trees rooted at A.



A shift-reduce parser therefore works

1. Bottom-up, grouping subtrees when reducing; and
2. Subtrees of a rule are formed from left-to-right - *think about this!*

This is equivalent to a rightmost derivation, *in reverse*.

Announcements (Monday, January 21st)

Milestones

- Snow!
- Continue picking your group (3 recommended). Who doesn't have a group?
- Group signup sheet will be distributed soon
- **Add-drop:** Tomorrow!

Assignment 1

- Any questions?
- **Due**: Friday, January 25th 11:59 PM

Midterm

- **Date**: February 26th from 6:00 - 7:30 PM in McConnell 103/321

Shift-Reduce Magic

The magic of shift-reduce parsers is the decision to either *shift* or *reduce*. How do we decide?

Shift

Shifting takes a token from the input stream and places it on the stack.

- More symbols are needed before we can apply a rule.

Reduce

Reducing replaces (multiple) symbols on the stack with a single symbol according to the grammar.

- Enough symbols have been processed to group subtrees, and the next input token is not part of a larger rule.

Conflicts

Shift-reduce (and reduce-reduce) conflicts occur when there is more than one possible option. We will revisit this soon!

Shift-Reduce Internals

- Implemented as a stack of *states* (not symbols);
- A state represents *the top* contents of the stack, *without* having to scan the contents;
- Shift/reduce according to the current state, and the next *k* unprocessed tokens.
- *Note: this resembles a DFA with a stack!*

Standard Parser Driver

```
while not accepted do
    action = LookupAction(currentState, nextTokens)
    if action == shift<nextState>
        push(nextState)
    else if action == reduce<A->gamma>
        pop(|gamma|) // Each symbol in gamma pushed a state
        push(NextState(currentState, A))
done
```

Both actions change the state of the stack

- **Shift**: read the next input token, push a single state on a stack
- **Reduce**: replace all states pushed as part of γ with a new state for A on the stack

Example

Consider the previous grammar for a simple language with statements and expressions. Each grammar rule is given a number

0 $S' \rightarrow S\$$	3 $S \rightarrow$ print (L)	6 $E \rightarrow E$ + E	9 $L \rightarrow L$, E
1 $S \rightarrow S$; S	4 $E \rightarrow$ id	7 $E \rightarrow$ (S , E)	
2 $S \rightarrow$ id := E	5 $E \rightarrow$ num	8 $L \rightarrow E$	

Parsing internals

- The possible states of the parser (states on the stack) are represented in a DFA;
- Start with the initial state (s1) on the stack;
- Choose the next action using the state transitions;
- The actions are summarized in a table, indexed with (currentState, nextTokens):
  - **Shift(n)**: skip next input symbol and push state n
  - **Reduce(k)**: rule k is $A \rightarrow \gamma$; pop $|\gamma|$ times; lookup(stack top, A) in table
  - **Goto(n)**: push state n
  - **Accept**: report success

Example - Table

DFA state	id	num	print	;	,	+	:=	(	)	$	S	E	L
	terminals										non-terminals		
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9												g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

Error transitions are omitted in tables.

Example

s_1	a := 7$
shift(4)	
$s_1\ s_4$	:= 7$
shift(6)	
$s_1\ s_4\ s_6$	7$
shift(10)	
$s_1\ s_4\ s_6\ s_{10}$	$
reduce(5): $E \longrightarrow$ num	
$s_1\ s_4\ s_6\ \cancel{s_{10}}$	$
lookup(s_6,E) = goto(11)	
$s_1\ s_4\ s_6\ s_{11}$	$
reduce(2): $S \longrightarrow$ id := E	
$s_1\ \cancel{s_4}\ \cancel{s_6}\ \cancel{s_{11}}$	$
lookup(s_1,S) = goto(2)	
$s_1\ s_2$	$
accept	

LR(1) Parser

LR(1) is an algorithm that attempts to construct a parsing table from a grammar using

- *Left-to-right parse*;
- *Rightmost-derivation*; and
- *1 symbol lookahead*.

If no conflicts arise (shift/reduce, reduce/reduce), then we are happy; otherwise, fix the grammar!

Overall idea

1. Construct an NFA for the grammar;
   - Represent possible parse states for all grammar rules (i.e. the stack contents);
   - Use transitions between states as actions are applied;
2. Convert the NFA to a DFA using a powerset construction; and
3. Represent the DFA using a table.

LR(1) Items

An LR(1) item $A \rightarrow \alpha \, . \, \beta \quad x$ consists of

1. A grammar production, $A \rightarrow \alpha\beta$;
2. The RHS position, represented by '.'; and
3. A lookahead symbol, x.

An LR(1) item intuitively represents how much of a rule we have recognized so far (by the '.' placement).

- The sequence α is on top of the stack; and
- The head of the input is derivable from βx.

The lookahead symbol is thus the terminal required to *end* (apply) the rule once β has been processed.

An LR(1) state is a set of LR(1) items.

LR(1) NFA

The LR(1) NFA is constructed in stages, beginning with an item representing the start state

$S \rightarrow .\ A\$$	?

This LR item indicates a state where

- We are at the beginning of the rule;
- The next sequence of symbols will be derived from non-terminal A; and
- The lookahead symbol is empty - we can apply at the end of input.

From here, we add successors recursively until termination (no more expansion possible).

Let `FIRST(`A`)` be the set of terminals that can begin an expansion of non-terminal A.

Let `FOLLOW(`A`)` be the set of terminals that can follow an expansion of non-terminal A.

LR(1) NFA - Non-Terminals

Given the LR item below, we add two types of successors (states connected through transitions)

$A \rightarrow \alpha \, . \, B \, \beta$	x

ϵ successors

For each production of B, add ϵ successor (transition with ϵ)

$B \rightarrow \, . \, \gamma$	y

for each y $\in$ `FIRST(`β`x)`. Note the inclusion of x, which handles the case where β is nullable.

B-successor

We also add B-successor to be followed when a sequence of symbols is reduced to B.

$A \rightarrow \alpha \, B \, . \, \beta$	x

LR(1) NFA - Terminals

For the case where the symbol after the '.' is a terminal

$A \to \alpha$. y β	x

there is a single y-successor of the form

$A \to \alpha$ y . β	x

which corresponds to the input of the next part of the rule (y).

LR(1) Table Construction

The LR(1) table construction is based on the LR(1) DFA, “inlining” ϵ-transitions. If you follow other resources online this DFA is sometimes constructed directly using the closure of item sets.

For each LR(1) item in state k, we add the following entries to the parser table depending on the contents of β and the state s of the successor.

$A \rightarrow \alpha \; . \; \beta$	x

1. **Goto**(s): β is a non-terminal
2. **Shift**(s): β is a terminal
3. **Reduce**(r): β is empty (where r is the number of the rule)
4. **Accept**: we have $A \rightarrow B$. $

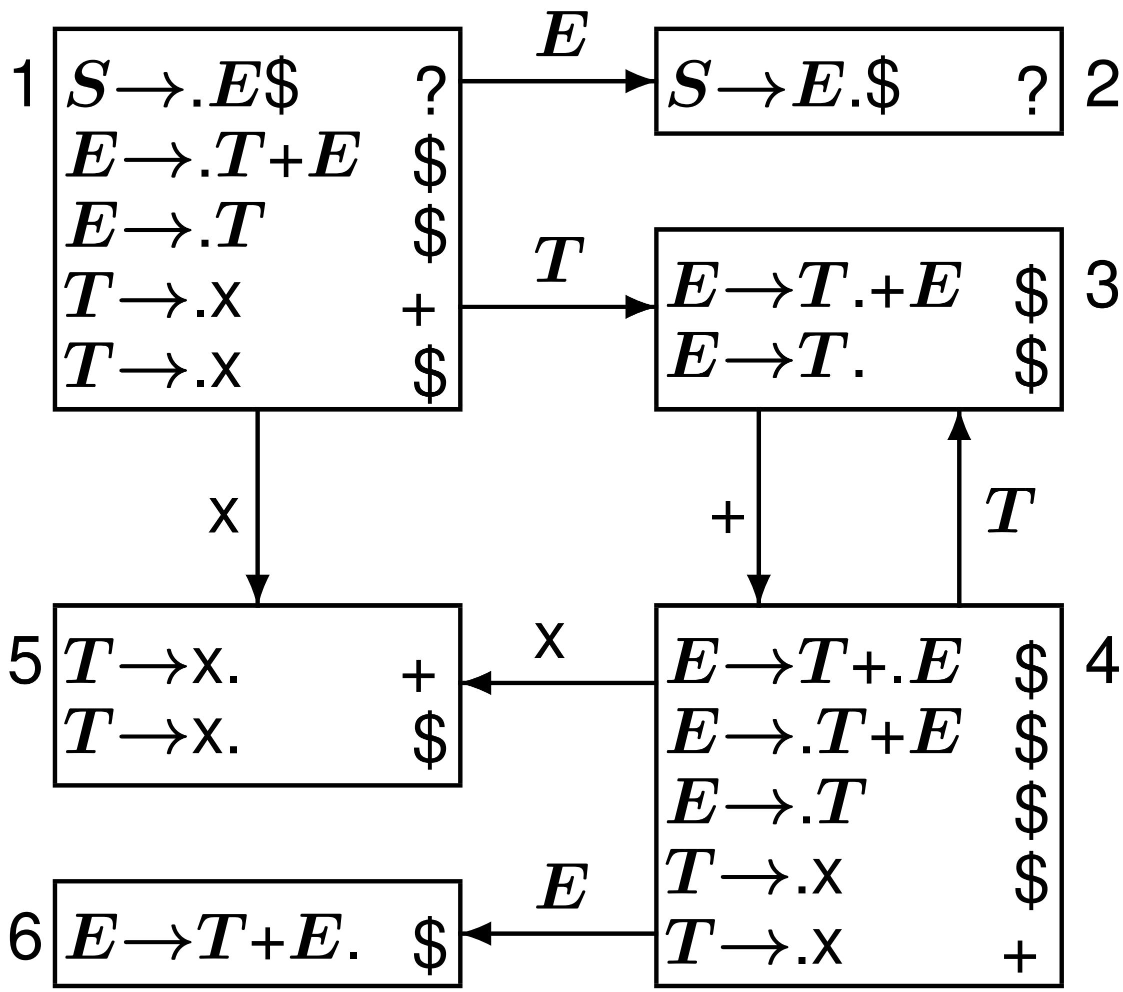
The next slide shows the construction of a simple expression grammar

$_0\; S \rightarrow E\$$ $\qquad$ $_2\; E \rightarrow T$

$_1\; E \rightarrow T + E$ $\qquad$ $_3\; T \rightarrow \text{x}$

Constructing the LR(1) DFA and Parser Table

Standard power-set construction, “inlining” ϵ-transitions.



	x	+	$	E	T
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

Parsing Conflicts

Parsing conflicts occur when there is more than one possible action for the parser to take which still results in a valid parse tree.

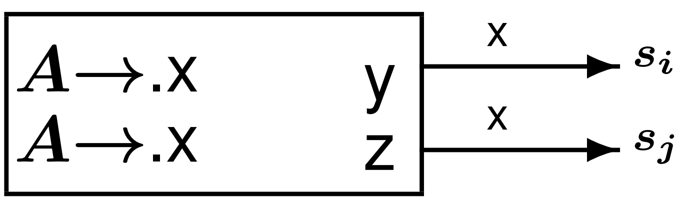
$A \rightarrow .x$ y
$A \rightarrow C.$ x

shift/reduce conflict

$A \rightarrow B.$ x
$A \rightarrow C.$ x

reduce/reduce conflict

What about shift/shift conflicts?



$\Rightarrow$ By construction of the DFA we have $s_i = s_j$

LALR Parsers

In practice, LR(1) tables may become very large for some programming languages. Parser generators use LALR(1), which merges states that are identical (same LR items) except for lookaheads. This introduces reduce/reduce conflicts.

Given the following example we begin by forming LR states

$S \rightarrow$ a E c $E \rightarrow$ e

$S \rightarrow$ a F d $F \rightarrow$ e

$S \rightarrow$ b F c

$S \rightarrow$ b E d

$E \rightarrow$ e.	c
$F \rightarrow$ e.	d

$E \rightarrow$ e.	d
$F \rightarrow$ e.	c

Since the states are identical other than lookahead, they are merged, introducing a reduce/reduce conflict.

$E \rightarrow$ e.	c,d
$F \rightarrow$ e.	c,d

`bison` Example

The grammar given below is expressed in `bison` as follows

$_1\; E \rightarrow \text{id}$ $\quad_3\; E \rightarrow E * E$ $\quad_5\; E \rightarrow E + E$ $\quad_7\; E \rightarrow (\, E\,)$

$_2\; E \rightarrow \text{num}$ $\quad_4\; E \rightarrow E \,/\, E$ $\quad_6\; E \rightarrow E - E$

```
%{
    /* C declarations */
%}

/* Bison declarations; tokens come from lexer (scanner) */
%token tIDENTIFIER tINTVAL

/* Grammar rules after the first %% */
%start exp
%%
exp : tIDENTIFIER
    | tINTVAL
    | exp '*' exp
    | exp '/' exp
    | exp '+' exp
    | exp '-' exp
    | '(' exp ')'
;
%% /* User C code after the second %% */
```

`bison` Example

For states which have no ambiguity, `bison` follows the idea we just presented. Using the `--verbose` option allows us to inspect the generated states and associated actions.

```
State 9
     5 exp: exp '+' . exp

     tIDENTIFIER  shift, and go to state 1
     tINTVAL      shift, and go to state 2
     '('          shift, and go to state 3

     exp  go to state 14

[...]

State 1
     1 exp: tIDENTIFIER .

     $default  reduce using rule 1 (exp)

State 2
     2 exp: tINTVAL .

     $default  reduce using rule 2 (exp)
```

`bison` Conflicts

As we previously discussed, the basic expression grammar is ambiguous.

`bison` reports cases where more than one parse tree is possible as `shift/reduce` or `reduce/reduce` conflicts.

```
$ bison --verbose tiny.y # --verbose produces tiny.output
tiny.y contains 16 shift/reduce conflicts.
```

Using the `--verbose` option we can output a full diagnostics log

```
$ cat tiny.output
State 12 contains 4 shift/reduce conflicts.
State 13 contains 4 shift/reduce conflicts.
State 14 contains 4 shift/reduce conflicts.
State 15 contains 4 shift/reduce conflicts.

[...]
```

`bison` Conflicts

Examining `State 14`, we see that the parser may reduce using rule ($E \rightarrow E + E$) or shift. This corresponds to grammar ambiguity, where the parser must choose between 2 different parse trees.

```
3 exp: exp . '*' exp
4    | exp . '/' exp
5    | exp . '+' exp
5    | exp '+' exp .    <-- problem is here
6    | exp . '-' exp

'*'  shift, and go to state 7
'/'  shift, and go to state 8
'+'  shift, and go to state 9
'-'  shift, and go to state 10

'*'       [reduce using rule 5 (exp)]
'/'       [reduce using rule 5 (exp)]
'+'       [reduce using rule 5 (exp)]
'-'       [reduce using rule 5 (exp)]
$default  reduce using rule 5 (exp)
```

`bison` Resolving Conflicts (Rewriting)

The first option in `bison` involves rewriting the grammar to resolve ambiguities (terms/factors)

$E \rightarrow E + T$	$T \rightarrow T * F$	$F \rightarrow$ id
$E \rightarrow E$ - T	$T \rightarrow T \,/\, F$	$F \rightarrow$ num
$E \rightarrow T$	$T \rightarrow F$	$F \rightarrow (E)$

```
%token tIDENTIFIER tINTVAL

%start exp
%%
exp : exp '+' term
    | exp '-' term
    | term
;
term : term '*' factor
     | term '/' factor
     | factor
;
factor : tIDENTIFIER
       | tINTVAL
       | '(' exp ')'
;
```

bison Resolving Conflicts (Directives)

bison also provides precedence directives which automatically resolve conflicts

```
%token tIDENTIFIER tINTVAL

%left '+' '-'    /* left-associative, lower precedence */
%left '*' '/'    /* left-associative, higher precedence */

%start exp
%%
exp : tIDENTIFIER
    | tINTVAL
    | exp '*' exp
    | exp '/' exp
    | exp '+' exp
    | exp '-' exp
    | '(' exp ')'
;
```

`bison` Resolving Conflicts (Directives)

The conflicts are automatically resolved using either shifts or reduces depending on the directive.

```
Conflict in state 11 between rule 5 and token '+'
        resolved as reduce. <-- Reduce exp + exp . +
Conflict in state 11 between rule 5 and token '-'
        resolved as reduce. <-- Reduce exp + exp . -
Conflict in state 11 between rule 5 and token '*'
        resolved as shift.  <-- Shift exp + exp . *
Conflict in state 11 between rule 5 and token '/'
        resolved as shift.  <-- Shift exp + exp . /
```

Note that this is not the same state 11 as before

Observations

- For operations with the same precedence and left associativity, we prefer reducing
- When the reduction contains an operation of lower precedence than the lookahead token, we prefer shifting

`bison` Resolving Conflicts (Directives)

- `%left` *(left-associative)*
- `%right` *(right-associative)*
- `%nonassoc` *(non-associative)*

Precedences are ordered from lowest to highest on a linewise basis.

When constructing a parse table, the action is chosen based on the precedence of the lookahead symbol and the symbol in the reduction. *Note this is much more general than expressions.*

If precedences are equal, then

- `%left`: favors reducing
- `%right`: favors shifting
- `%nonassoc`: yields an error

This *usually* ends up working.

The Dangling Else Problem - LR

The following 2 slides have been adapted from "Modern Compiler Implementation in Java", by Appel and Palsberg.

$P \rightarrow L$

$L \rightarrow S$

$L \rightarrow L; S$

$S \rightarrow$ "while" ident "do" S

$S \rightarrow$ "if" ident "then" S

$S \rightarrow$ "if" ident "then" S "else" S

$S \rightarrow$ ident := ident

$S \rightarrow$ "begin" L "end"

The Dangling Else Problem - LR

Solving the dangling else ambiguity in LR parsers requires differentiating between *matched* and *unmatched* statements.

$S \rightarrow$ "while" ident "do" S

$S \rightarrow$ "if" ident "then" S

$S \rightarrow$ "if" ident "then" S_{matched} "else" S

$S \rightarrow S_{\text{no trailing}}$

$S_{\text{no trailing}} \rightarrow$ ident := ident

$S_{\text{no trailing}} \rightarrow$ "begin" L "end"

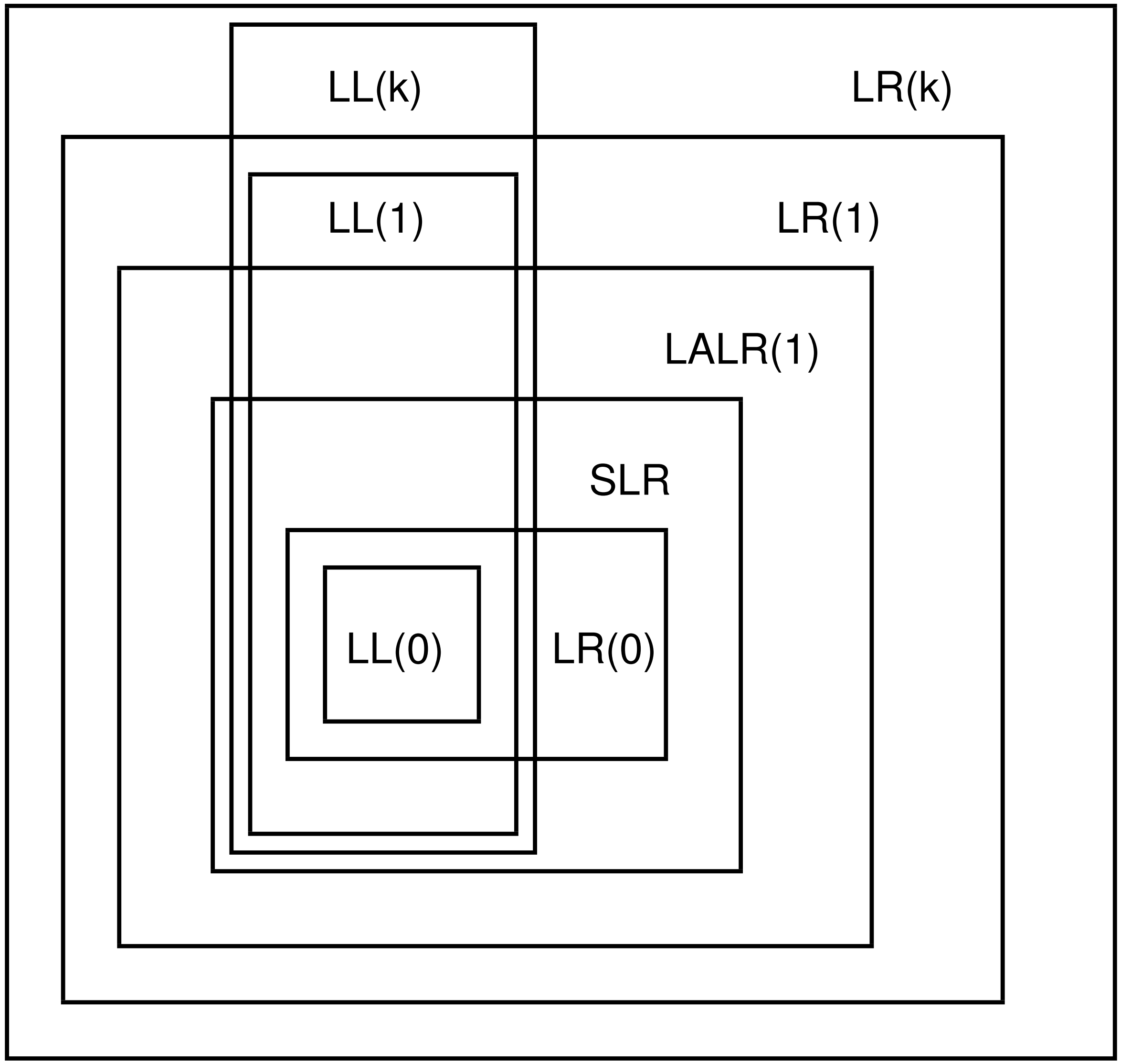
$S_{\text{matched}} \rightarrow S_{\text{no trailing}}$

$S_{\text{matched}} \rightarrow$ "while" ident "do" S_{matched}

$S_{\text{matched}} \rightarrow$ "if" ident "then" S_{matched} "else" S_{matched}

Since we match to the nearest *unmatched* if-statement, a *matched* if-statement **cannot** have any unmatched statements nested (or this breaks the condition)

Comparison of Languages Accepted by Parser Generators



Takeaways

What you should know

- What it means to shift and reduce;
- Shift/reduce that can occur in LR parsers and how to resolve them; and
- The *general* idea of the LR states at a *high-level*;

What you do not need to know

- Building a parser DFA/NFA/Table (you should understand how to use them though);
- Detailed understanding of LL/LR internals (e.g. `FIRST` and `FOLLOW` sets); and
- LALR parsers;

For this class you should focus on intuition and practice rather than memorizing exact definitions and algorithms.