

Parsing

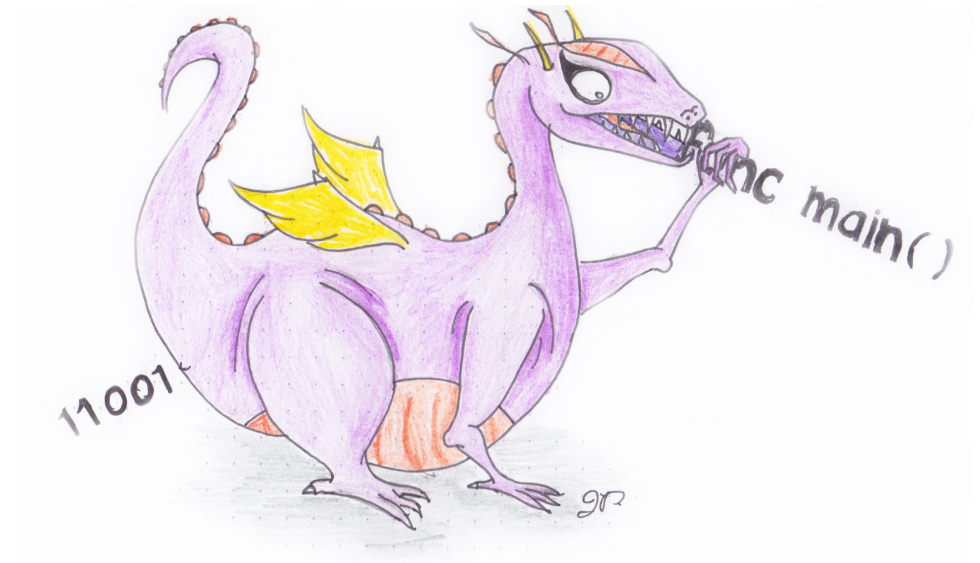
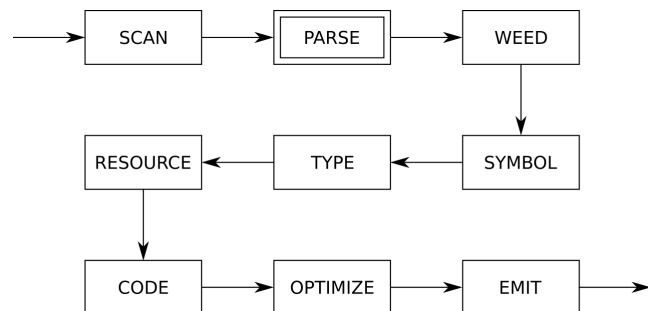
COMP 520: Compiler Design (4 credits)

Alexander Krolik

`alexander.krolik@mail.mcgill.ca`

MWF 9:30-10:30, TR 1080

`http://www.cs.mcgill.ca/~cs520/2018/`



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/~cs520/2018/>)

- flex, bison, and/or SableCC

Announcements (Monday, January 15th)

Milestones

- Continue picking your group (3 recommended). Who doesn't have a group?
- Learn `flex/bison` or `SableCC` – Assignment 1 out today!
- Say hi to the Go Gopher!
- Facebook group: <https://www.facebook.com/groups/1588690564524014/>

Midterm

- 1.5 hour “in class” midterm, so either 30 minutes before/after class. *Thoughts?*
- **Tentative date:** Friday, March 16th. *Thoughts?*

Parsing

The parsing phase of a compiler

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

Internally

- It corresponds to a *deterministic push-down 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;
- Allows recognizing a larger set of languages to DFAs/NFAs;
- Has a stack that can be viewed/manipulated by transitions; and
- Is used to recognize a context-free language.

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

Context-Free Grammars

In the language hierarchy, context-free grammars

- Are stronger than regular expressions;
- Generate context-free languages; and
- Are able to express recursively-defined constructs.

Example: Try writing a regular expression for parentheses matching (where the number of parentheses is unbounded)

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

The solution using a CFG is simple

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

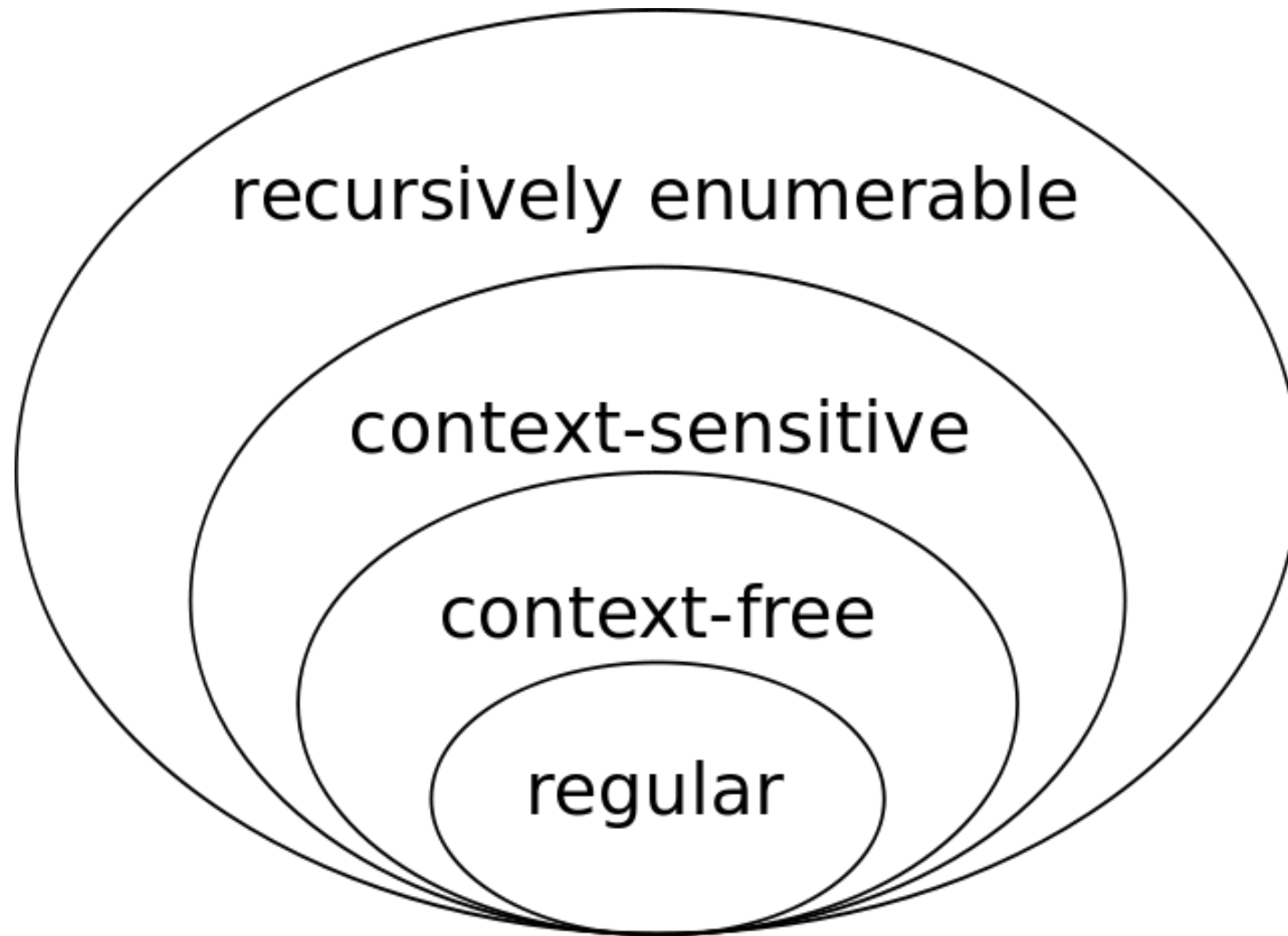
Notes on Context-Free Languages

- 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:

$$\{a^n b^n c^n \mid n \geq 1\}$$

- 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* push-down automaton (same idea as DFA vs NFA, only one transition possible from a given state).
 - DPDAs cannot recognize all context-free languages!
 - **Example:** even length palindrome $E \rightarrow a E a \mid b E b \mid \epsilon$. How do we know that matching should start?

Chomsky Hierarchy



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

Example CFG

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

Simple CFG

$$A \rightarrow a B$$

$$A \rightarrow \epsilon$$

$$B \rightarrow b B$$

$$B \rightarrow c$$

Alternatively

$$A \rightarrow a B \mid \epsilon$$

$$B \rightarrow b B \mid c$$

In both cases we specify $S = A$. Can you write this grammar as a regular expression?

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 , $abbc$, $abbbc$, \dots

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 A

$$A \rightarrow A A \mid B \mid a$$

$$B \rightarrow b B \mid c$$

A

A A

A B

a B

$a b$ B

$a b c$

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 A

$$A \rightarrow A A \mid B \mid a$$

$$B \rightarrow b B \mid c$$

Rightmost

A

A A

A B

A b B

A b c

a b c

Leftmost

A

A A

a A

a B

a b B

a b c

Example Programming Language

CFG rules

$\text{Prog} \rightarrow \text{Dcls Stmt}$

$\text{Dcls} \rightarrow \text{Dcl Dcls} \mid \epsilon$

$\text{Dcl} \rightarrow \text{"int" ident} \mid \text{"float" ident}$

$\text{Stmts} \rightarrow \text{Stmt Stmts} \mid \epsilon$

$\text{Stmt} \rightarrow \text{ident "=" Val}$

$\text{Val} \rightarrow \text{num} \mid \text{ident}$

Corresponding Program

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

Leftmost derivation

Prog

Dcls *Stmts*

Dcl *Dcls Stmts*

"int" ident *Dcls* *Stmts*

"int" ident "float" ident *Dcls* *Stmts*

"int" ident "float" ident *Stmts*

"int" ident "float" ident *Stmt* *Stmts*

"int" ident "float" ident ident "=" *Val* *Stmts*

"int" ident "float" ident ident "=" ident *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
$A \rightarrow A a \mid b$ (left-recursive)	b	$\underline{A} a$ $\underline{A} a a$ $b a a$	$A \rightarrow b \{ a \}$
$A \rightarrow a A \mid b$ (right-recursive)	b	$a \underline{A}$ $a a \underline{A}$ $a a b$	$A \rightarrow \{ a \} b$

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

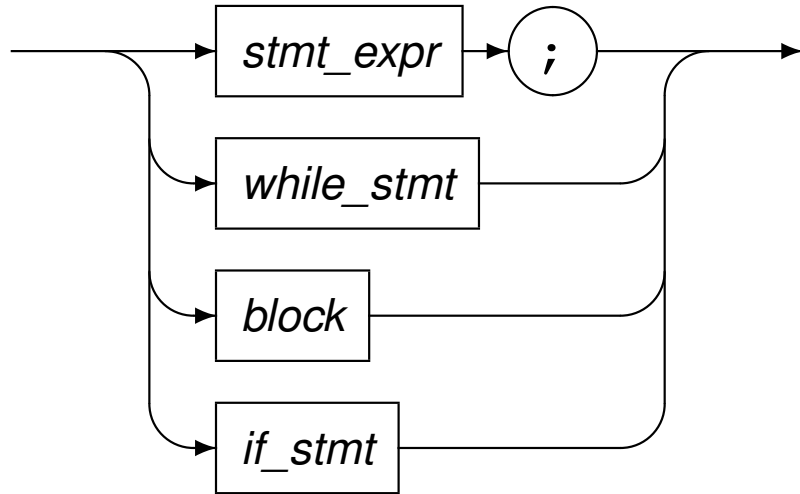
```
if_stmt ::= IF "(" expr ")" stmt |  
           IF "(" expr ")" stmt ELSE stmt
```

can be simplified and re-written as

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

Railroad Diagrams (thanks rail.sty!)

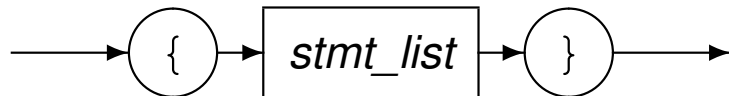
stmt



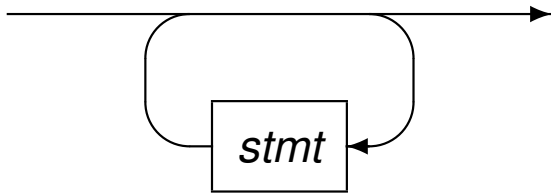
while_stmt



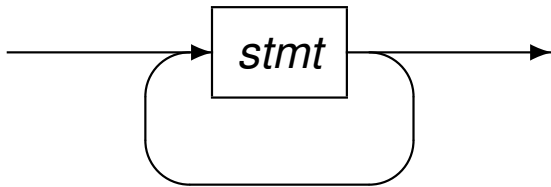
block

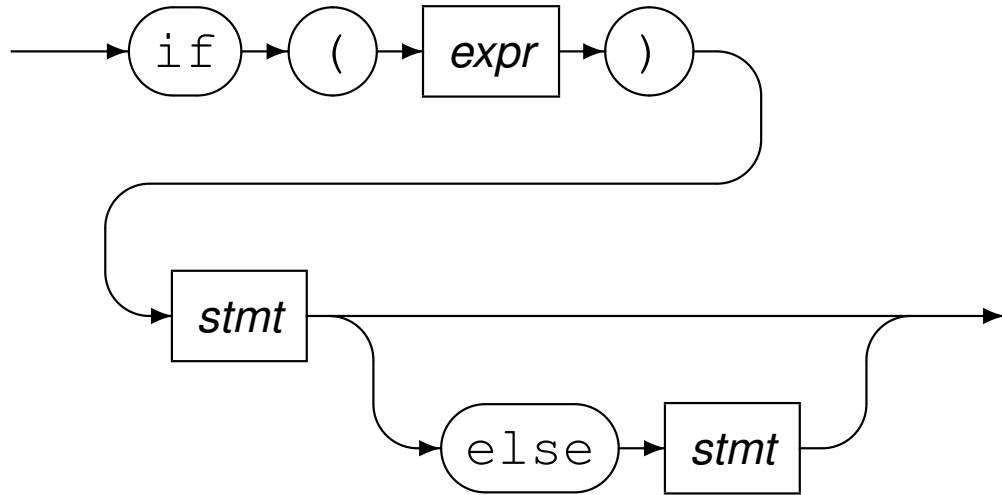


stmt_list (0 or more)



stmt_list (1 or more)



if_stmt

Announcements (Wednesday, January 17th)

Milestones

- Continue picking your group (3 recommended). Who doesn't have a group?
- Facebook group: <https://www.facebook.com/groups/1588690564524014/>
- Office hours
 - Alex: Wednesdays 10:30-11:30
 - David: Thursdays 11:30-12:30
 - Email/Facebook if these times don't work!

Assignment 1

- Any questions?
- **Due:** Sunday, January 28th 11:59 PM

Midterm

- 1.5 hour “in class” midterm, so either 30 minutes before/after class. *Thoughts?*
- **Tentative date:** Friday, March 16th. *Thoughts?*

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

Should be online by tomorrow

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; and
- The exact structure of the tree depends on the order of the derivation (rightmost and leftmost derivations generate different parse trees!).

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

Example

Grammar

$$\begin{array}{lll}
 S \rightarrow S ; S & E \rightarrow \text{id} & L \rightarrow E \\
 S \rightarrow \text{id} := E & E \rightarrow \text{num} & L \rightarrow L , E \\
 S \rightarrow \text{print} (L) & E \rightarrow E + E & \\
 & E \rightarrow (S , E) &
 \end{array}$$

Derive the following program using the above grammar

```

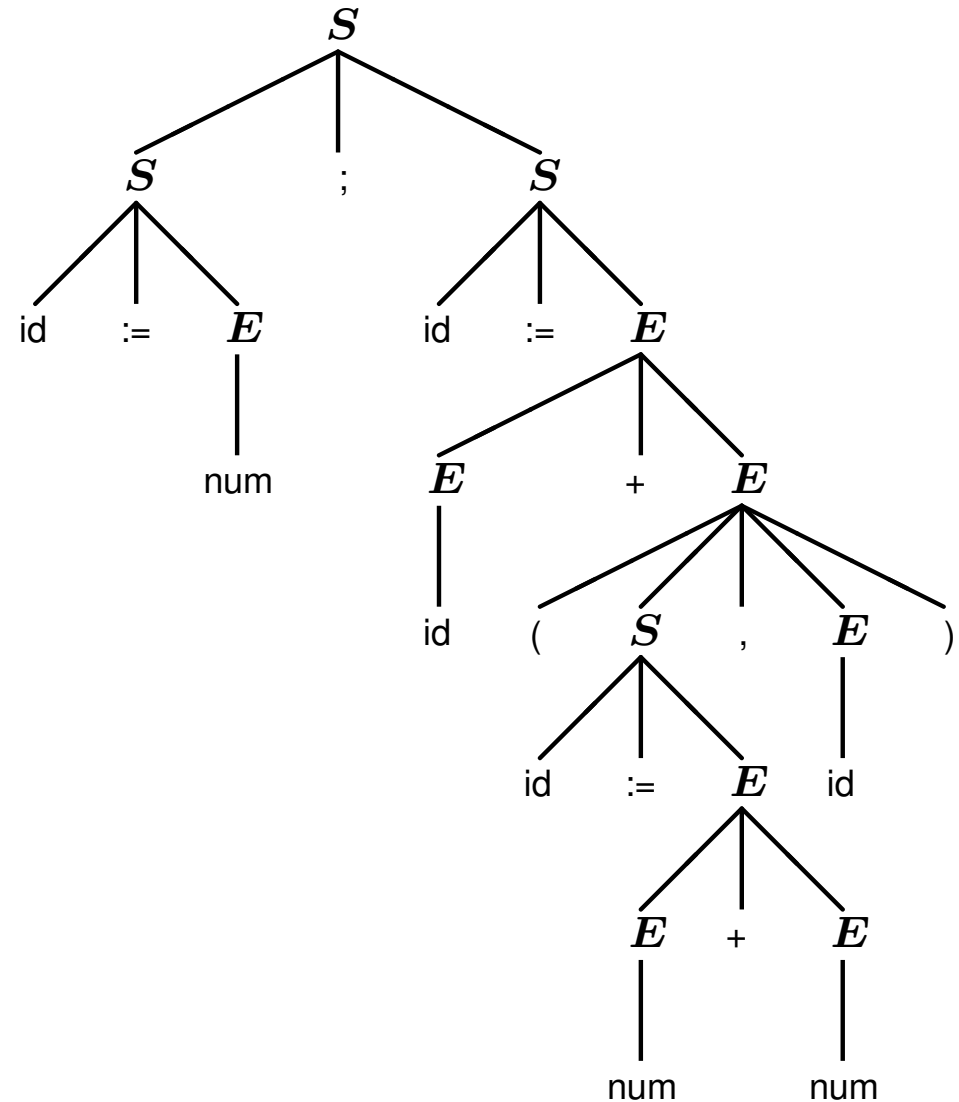
a  := 7;
b  := c + (d := 5 + 6, d)

```

Rightmost derivation

$$\begin{array}{l}
 \underline{S} \\
 S; \underline{S} \\
 S; \text{id} := \underline{E} \\
 S; \text{id} := E + \underline{E} \\
 S; \text{id} := E + (S, \underline{E}) \\
 S; \text{id} := E + (\underline{S}, \text{id}) \\
 S; \text{id} := E + (\text{id} := \underline{E}, \text{id}) \\
 S; \text{id} := E + (\text{id} := E + \underline{E}, \text{id}) \\
 S; \text{id} := E + (\text{id} := \underline{E} + \text{num}, \text{id}) \\
 S; \text{id} := \underline{E} + (\text{id} := \text{num} + \text{num}, \text{id}) \\
 \underline{S}; \text{id} := \text{id} + (\text{id} := \text{num} + \text{num}, \text{id}) \\
 \text{id} := \underline{E}; \text{id} := \text{id} + (\text{id} := \text{num} + \text{num}, \text{id}) \\
 \text{id} := \text{num}; \text{id} := \text{id} + (\text{id} := \text{num} + \text{num}, \text{id})
 \end{array}$$

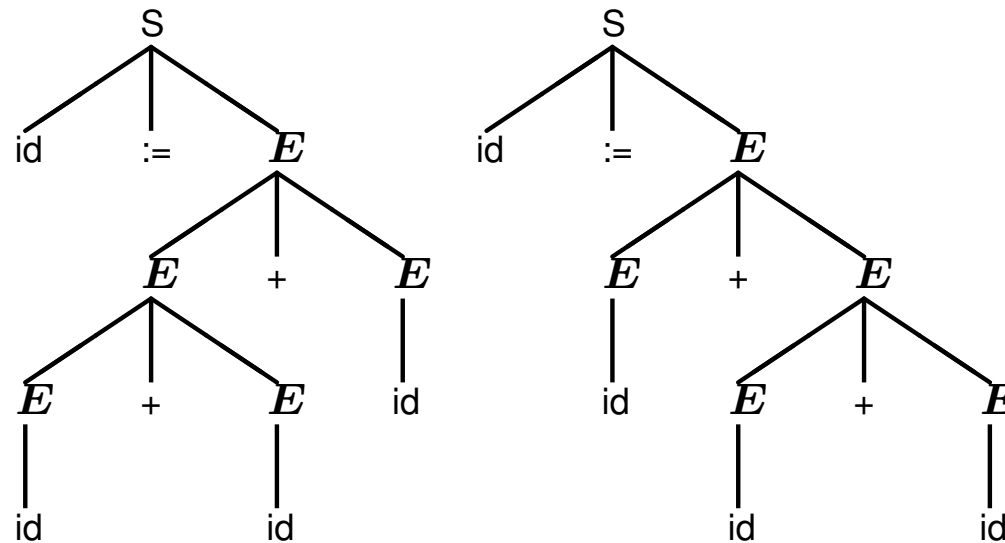
Rightmost derivation

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


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

Clearly, ambiguous grammars can have potentially severe consequences. How can we make grammars unambiguous?

- Note that not all context-free languages have an unambiguous grammar (COMP 330);
- However, deterministic pushdown automata that are used by parsers *require* an unambiguous grammar; so
- In practice, we either 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 an ambiguous grammar for expressions (refer to the previous slides to understand this ambiguity)

$$E \rightarrow \text{id} \quad E \rightarrow E / E \quad E \rightarrow (E)$$

$$E \rightarrow \text{num} \quad E \rightarrow E + E$$

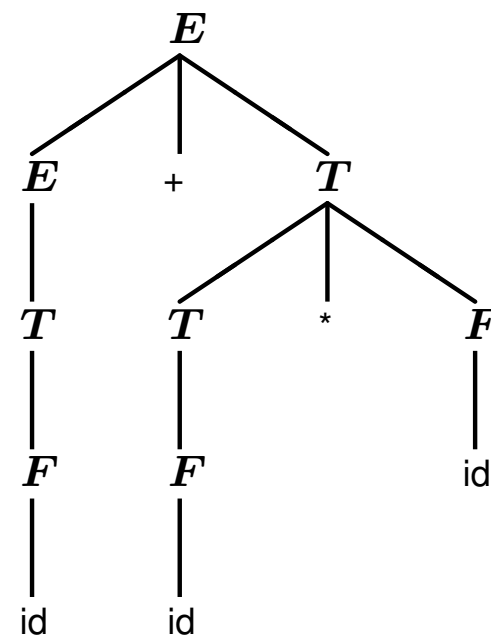
$$E \rightarrow E * E \quad E \rightarrow E - E$$

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

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

$$E \rightarrow E - T \quad T \rightarrow T / F \quad F \rightarrow \text{num}$$

$$E \rightarrow T \quad T \rightarrow F \quad F \rightarrow (E)$$



Why does this work?

Rewriting an Ambiguous Expression Grammar

Expression grammars must have 2 mathematical attributes for operations

- **Precedence:** Order of operations ($*$ and $/$ have precedence 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

$$\begin{aligned} \text{IfStmt} \rightarrow & \text{tIF Expr tTHEN Stmt tELSE Stmt} \\ & | \text{tIF Expr tTHEN Stmt} \end{aligned}$$

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

if {expr} then	tIF
if {expr} then	EXPR
<stmt>	tTHEN
else	tIF
<stmt>	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

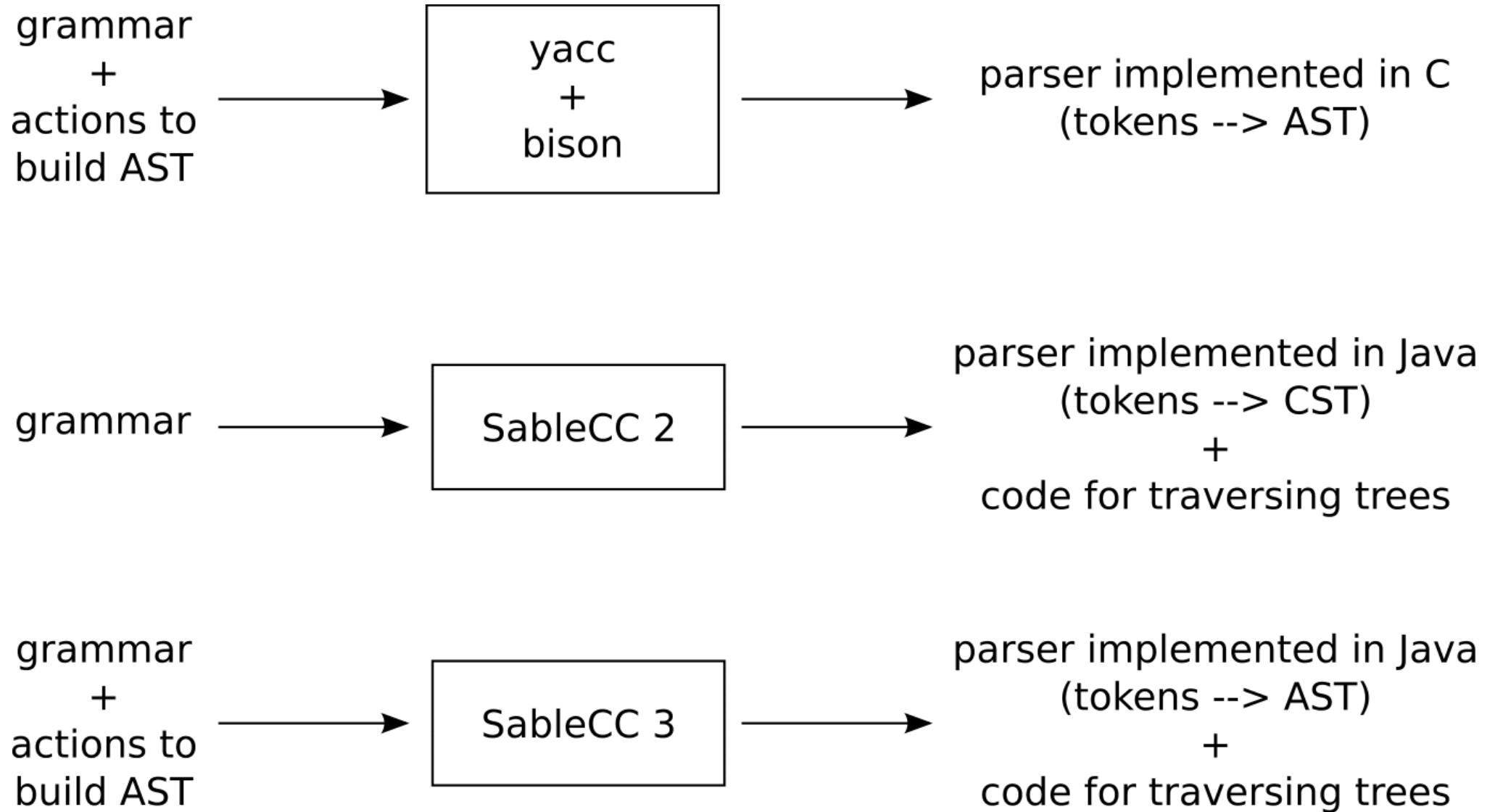
1. Top-down, *predictive* or *recursive descent* parsers. Used in all languages designed by Wirth, e.g. Pascal, Modula, and Oberon; and
2. Bottom-up parsers.

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`) is a parser generator

- 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.

Example `bison` File

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

$_1 E \rightarrow \text{id}$ $_3 E \rightarrow E * E$ $_5 E \rightarrow E + E$ $_7 E \rightarrow (E)$

$_2 E \rightarrow \text{num}$ $_4 E \rightarrow E / E$ $_6 E \rightarrow E - E$

```
%{
    /* 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 **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
exp.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 **Conflicts**

Examining state 11, we see that the parser reports that it may reduce using rule 3 ($E \rightarrow E * E$) or shift.

This corresponds to grammar ambiguities, where the parser has a choice between 2 different parse trees.

```
exp  ->  exp . '*' exp      (rule 3)
exp  ->  exp '*' exp .      (rule 3)  <-- problem is here
exp  ->  exp . '/' exp      (rule 4)
exp  ->  exp . '+' exp      (rule 5)
exp  ->  exp . '-' exp      (rule 6)
```

```
'*'      shift, and go to state 6
'/'      shift, and go to state 7
'+'      shift, and go to state 8
'-'      shift, and go to state 9
```

```
'*'      [reduce using rule 3 (exp)]
'/'      [reduce using rule 3 (exp)]
'+'      [reduce using rule 3 (exp)]
'-'      [reduce using rule 3 (exp)]
$default reduce using rule 3 (exp)
```

bison Resolving Conflicts (Rewriting)

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

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

$$E \rightarrow E - T \quad T \rightarrow T / F \quad F \rightarrow \text{num}$$

$$E \rightarrow T \quad T \rightarrow F \quad F \rightarrow (E)$$

```
%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, 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.

We'll come back to the following later!

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.

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

Unary Minus

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

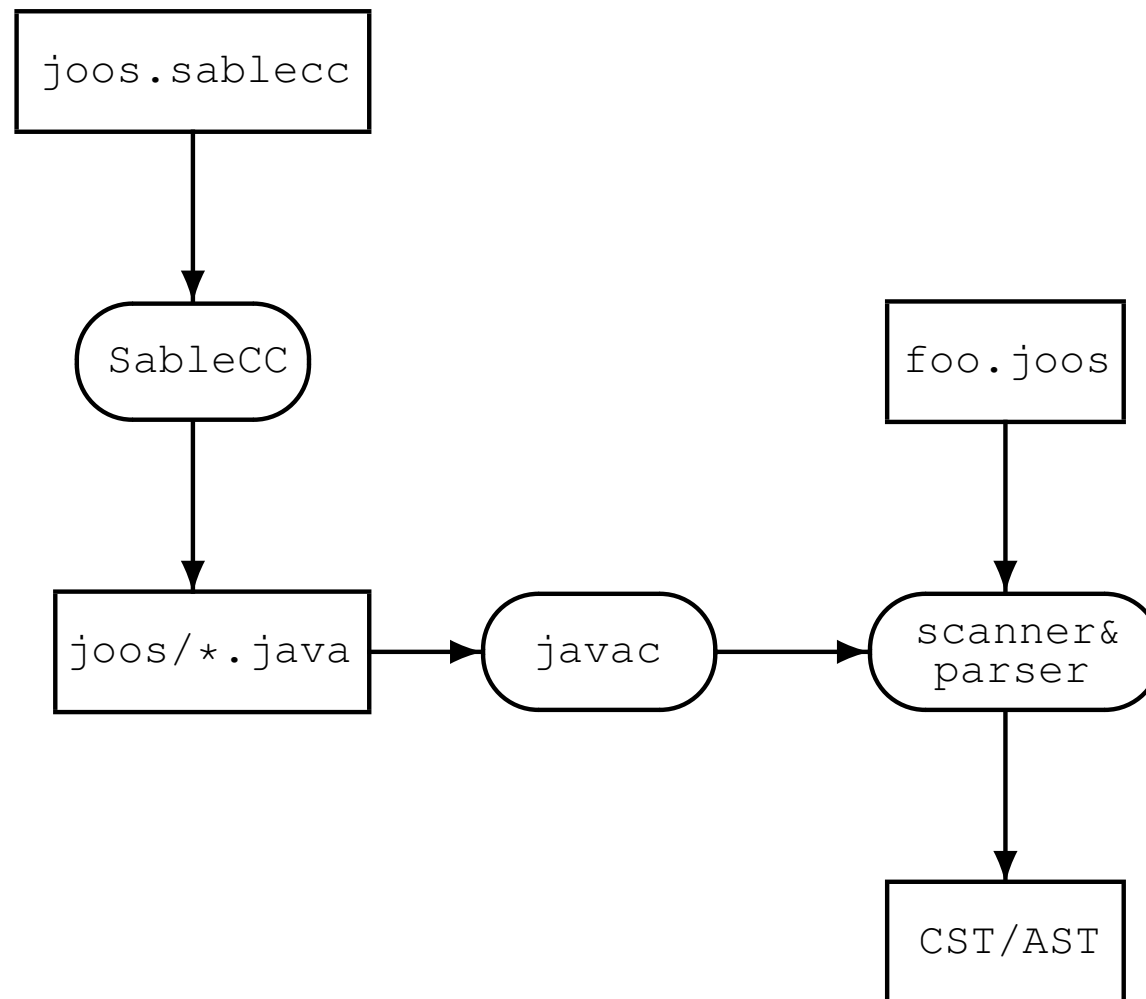
To encourage `bison` to behave as expected, we use precedence directives with a special unused token

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

```
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

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 19th)

Milestones

- Continue picking your group (3 recommended). Who doesn't have a group?
- Facebook group: <https://www.facebook.com/groups/1588690564524014/>
- Office hours
 - Alex: Wednesdays 10:30-11:30
 - David: Thursdays 11:30-12:30

Assignment 1

- Reference compiler has been posted
- Any questions?
- **Due:** Sunday, January 28th 11:59 PM

Midterm

- **Tentative:** Week of Monday, March 12th, 1.5 hour “evening” midterm. *Thoughts?*

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

$\text{Prog} \rightarrow \text{Dcls Stmt}$

$\text{Dcls} \rightarrow \text{Dcl Dcls} \mid \epsilon$

$\text{Dcl} \rightarrow \text{"int" ident} \mid \text{"float" ident}$

$\text{Stmts} \rightarrow \text{Stmt Stmts} \mid \epsilon$

$\text{Stmt} \rightarrow \text{ident "=" Val}$

$\text{Val} \rightarrow \text{num} \mid \text{ident}$

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

Prog

Dcls Stmts

Dcl Dcls Stmts

“int” ident Dcls Stmts

“int” ident Dcl Dcls Stmts

“int” ident “float” ident Dcls Stmts

“int” ident “float” ident Stmts

“int” ident “float” ident Stmt Stmts

“int” ident “float” ident ident “=” Val Stmts

“int” ident “float” ident ident “=” ident Stmts

“int” ident “float” ident ident “=” ident

Next Token

t INT

t INT

t INT

t FLOAT

t FLOAT

t IDENTIFIER

t IDENTIFIER

t IDENTIFIER

t IDENTIFIER

EOF

Options

Dcls Stmts

Dcl Dcls | ϵ

“int” ident | “float” ident

Dcl Dcls | ϵ

“int” ident | “float” ident

Dcl Dcls | ϵ

Stmt Stmts | ϵ

ident “=” Val

num | ident

Stmt Stmts | ϵ

Recursive Descent Parsers

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

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

- Each non-terminal has a *predict set* for each of its rules that indicates if the rule can be applied given the k lookahead tokens; and
- If the next token is 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

$$\text{Prog} \rightarrow \text{Dcls Stmts}$$
$$\text{Dcls} \rightarrow \text{Dcl Dcls} \mid \epsilon$$
$$\text{Dcl} \rightarrow \text{"int" ident} \mid \text{"float" ident}$$

We have 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:
        /* no more declarations, parsing
        continues in the Prog method */
        return
    end
end
```

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

Common Prefixes

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

$$\text{IfStmt} \rightarrow \text{tIF Exp tTHEN Stmts tEND} \mid \text{tIF Exp tTHEN Stmts tELSE Stmts tEND}$$

With a single token of lookahead (an LL(1) parser), we are unable to predict which rule to follow (both rules have the token `tIF` in their predict set).

Solution

To resolve this issue, we *factor* the grammar

$$\text{IfStmt} \rightarrow \text{tIF Exp tTHEN Stmts IfEnd}$$
$$\text{IfEnd} \rightarrow \text{tEND} \mid \text{tELSE Stmts tEND}$$

There is now only a single IfStmt rule and thus there is no ambiguity and the grammar is LL(1). Each production for the IfEnd variable has a different (non-intersecting) predict set

1. `tEND`
2. `tELSE`

Left Recursion

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

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

$$\beta \rightarrow \texttt{t} \text{ TOKEN}$$

Assume we can come up with a predict set for A consisting of $\texttt{t} \text{ TOKEN}$, then applying this rule gives us

Expansion	Next Token
<u>A</u>	$\texttt{t} \text{ TOKEN}$
<u>A</u> β	$\texttt{t} \text{ TOKEN}$
<u>A</u> β β	$\texttt{t} \text{ TOKEN}$
<u>A</u> β β β	$\texttt{t} \text{ TOKEN}$
<u>A</u> β β β β	$\texttt{t} \text{ TOKEN}$
<u>A</u> β β β β β	$\texttt{t} \text{ TOKEN}$
...	

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
```

Announcements (Monday, January 22nd)

Milestones

- Continue picking your group (3 recommended). Who doesn't have a group?
- Facebook group: <https://www.facebook.com/groups/1588690564524014/>
- Office hours
 - Alex: Wednesdays 10:30-11:30
 - David: Thursdays 11:30-12:30

Assignment 1

- Any questions?
- **Due:** Sunday, January 28th 11:59 PM

Midterm

- **Tentative:** Week of Monday, March 12th, 1.5 hour “evening” midterm. *Thoughts?*

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 symbol $\$$; 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.

$$\begin{array}{llll} S' \rightarrow S\$ & S \rightarrow S ; S & E \rightarrow \text{id} & L \rightarrow E \\ & S \rightarrow \text{id} := E & E \rightarrow \text{num} & L \rightarrow L , E \\ & S \rightarrow \text{print} (L) & E \rightarrow E + E & \\ & & E \rightarrow (S , E) & \end{array}$$

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 representing 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) \$	$E \rightarrow \text{num}$
id := E	; b := c + (d := 5 + 6 , d) \$	$S \rightarrow \text{id} := E$
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 := id + (d := 5 + 6 , d) \$	$E \rightarrow \text{id}$
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 := num + 6 , d) \$	$E \rightarrow \text{num}$
S ;	id := E + (id := E + 6 , d) \$	shift
S ;	id := E + (id := E + 6 , d) \$	shift
S ;	id := E + (id := E + num , d) \$	$E \rightarrow \text{num}$
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:

S

S; S

S; id := E

S; id := E + E

S; id := E + (S, E)

S; id := E + (S, id)

S; id := E + (id := E, id)

S; id := E + (id := E + E, id)

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

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

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

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

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

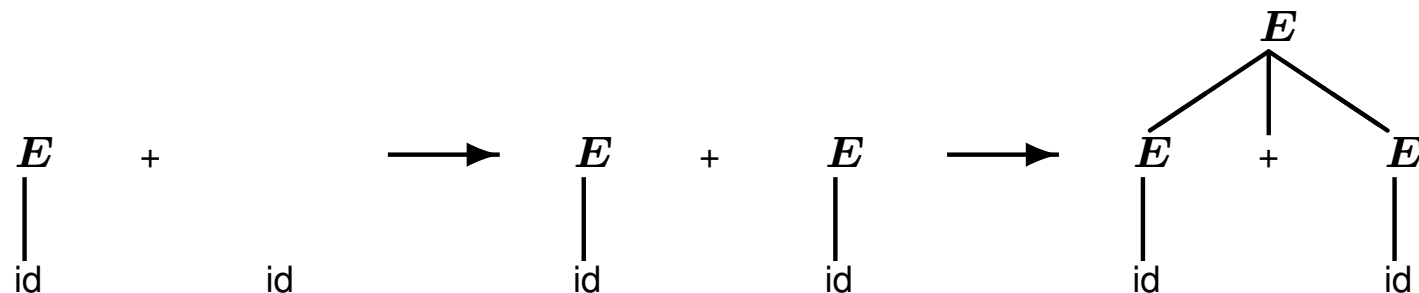
Note that the rules applied in LR parsing are the same as those above, *in reverse*.

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; 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*.

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 rules.

- 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

- Each rule is given a number

$$_0 S' \rightarrow S\$ \quad _3 S \rightarrow \text{print} (L) \quad _6 E \rightarrow E + E \quad _9 L \rightarrow L , E$$

$$_1 S \rightarrow S ; S \quad _4 E \rightarrow \text{id} \quad _7 E \rightarrow (S , E)$$

$$_2 S \rightarrow \text{id} := E \quad _5 E \rightarrow \text{num} \quad _8 L \rightarrow E$$

- Start with the initial state (s1) on the stack;
- Choose the next action using a DFA - the stack contains only DFA states now;
- 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	terminals							non-terminals		
state	id	num	print	;	,	+	:=	()	\$
1	s4		s7							
2					s3					a
3	s4		s7							
4						s6				
5				r1	r1					r1
6	s20	s10						s8		
7								s9		
8	s4		s7							
9										
10					r5	r5	r5			r5

DFA	terminals										non-terminals		
state	id	num	print	;	,	+	:=	()	\$	<i>S</i>	<i>E</i>	<i>L</i>
11						r2	r2	s16		r2			
12						s3	s18						
13						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): $E \rightarrow \text{num}$	
$s_1\ s_4\ s_6\ /\$10$	$\$$
lookup(s_6, E) = goto(11)	
$s_1\ s_4\ s_6\ s_{11}$	$\$$
reduce(2): $S \rightarrow \text{id} := E$	
$s_1\ /\$4\ /\$6\ /\$11$	$\$$
lookup(s_1, S) = goto(2)	
$s_1\ s_2$	$\$$
accept	

LR(1)

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

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

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

LR(1) Items

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

1. A grammar production, $A \rightarrow \alpha \beta \gamma$;
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 $\beta \gamma x$.

The lookahead symbol is thus the terminal required to *end* (apply) the rule once $\beta \gamma$ 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

$$S \rightarrow \cdot A \$ \quad ?$$

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 $\text{FIRST}(A)$ be the set of terminals that can begin an expansion of non-terminal A .

Let $\text{FOLLOW}(A)$ 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)

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

ϵ successors

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

$$B \rightarrow . \gamma \quad y$$

for each $y \in \text{FIRST}(\beta_x)$. 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 .

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

LR(1) NFA - Terminals

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

$$A \rightarrow \alpha . y \beta \quad x$$

there is a single y-successor of the form

$$A \rightarrow \alpha y . \beta \quad x$$

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
--------------------------------	-----

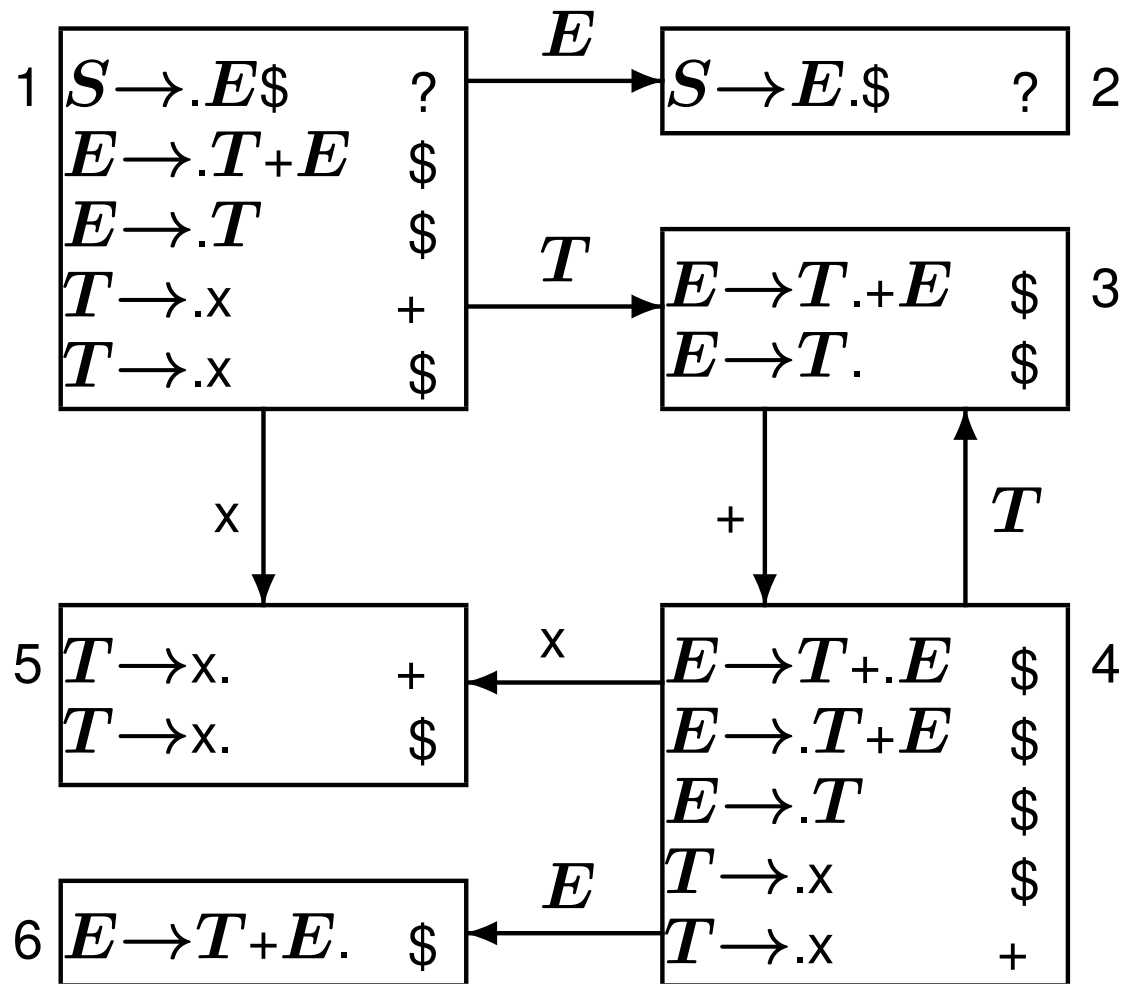
1. **Goto**(s): β is a non-terminal
2. **Shift**(s): β is a terminal
3. **Reduce**(r): β is empty (where r is the number of the rule)
4. **Accept**: we have $A \rightarrow B . \$$

The next slide shows the construction of a simple expression grammar

$$\begin{array}{ll}
 {}_0 S \rightarrow E\$ & {}_2 E \rightarrow T \\
 {}_1 E \rightarrow T + E & {}_3 T \rightarrow x
 \end{array}$$

Construing the LR(1) DFA and Parser Table

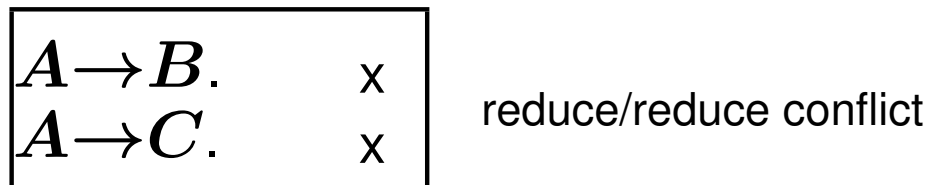
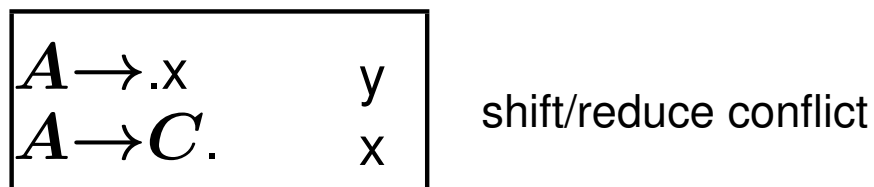
Standard power-set construction, “inlining” ϵ -transitions.



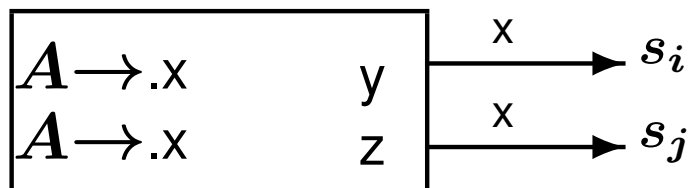
	x	+	\$	E	T
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

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.



What about shift/shift conflicts?



\Rightarrow By construction of the DF_A 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

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

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

$$S \rightarrow b F c$$

$$S \rightarrow b E d$$

$E \rightarrow e.$	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

Example `bison` File

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

$_1 E \rightarrow \text{id}$ $_3 E \rightarrow E * E$ $_5 E \rightarrow E + E$ $_7 E \rightarrow (E)$

$_2 E \rightarrow \text{num}$ $_4 E \rightarrow E / E$ $_6 E \rightarrow E - E$

```
%{
    /* 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 **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
exp.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 **Conflicts**

Examining state 11, we see that the parser reports that it may reduce using rule 3 ($E \rightarrow E * E$) or shift.

This corresponds to grammar ambiguities, where the parser has a choice between 2 different parse trees.

```
exp  ->  exp . '*' exp      (rule 3)
exp  ->  exp '*' exp .      (rule 3)  <-- problem is here
exp  ->  exp . '/' exp      (rule 4)
exp  ->  exp . '+' exp      (rule 5)
exp  ->  exp . '-' exp      (rule 6)
```

```
'*'      shift, and go to state 6
'/'      shift, and go to state 7
'+'      shift, and go to state 8
'-'      shift, and go to state 9
```

```
'*'      [reduce using rule 3 (exp)]
'/'      [reduce using rule 3 (exp)]
'+'      [reduce using rule 3 (exp)]
'-'      [reduce using rule 3 (exp)]
$default reduce using rule 3 (exp)
```

bison **Resolving Conflicts (Rewriting)**

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

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

$$E \rightarrow E - T \quad T \rightarrow T / F \quad F \rightarrow \text{num}$$

$$E \rightarrow T \quad T \rightarrow F \quad F \rightarrow (E)$$

```
%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, 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 3 slides have been adapted from "Modern Compiler Implementation in Java", by Appel and Palsberg.

$$P \rightarrow L$$

$$L \rightarrow S$$

$$L \rightarrow L; S$$

$$S \rightarrow \text{"while" ident "do" } S$$

$$S \rightarrow \text{"if" ident "then" } S$$

$$S \rightarrow \text{"if" ident "then" } S \text{ "else" } S$$

$$S \rightarrow \text{ident := ident}$$

$$S \rightarrow \text{"begin" } L \text{ "end"}$$

The Dangling Else Problem - LR

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

$$S \rightarrow \text{"while" ident "do" } S$$

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

$$S \rightarrow \text{"if" ident "then" } S$$

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

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

$$\text{"else" } S$$

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

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

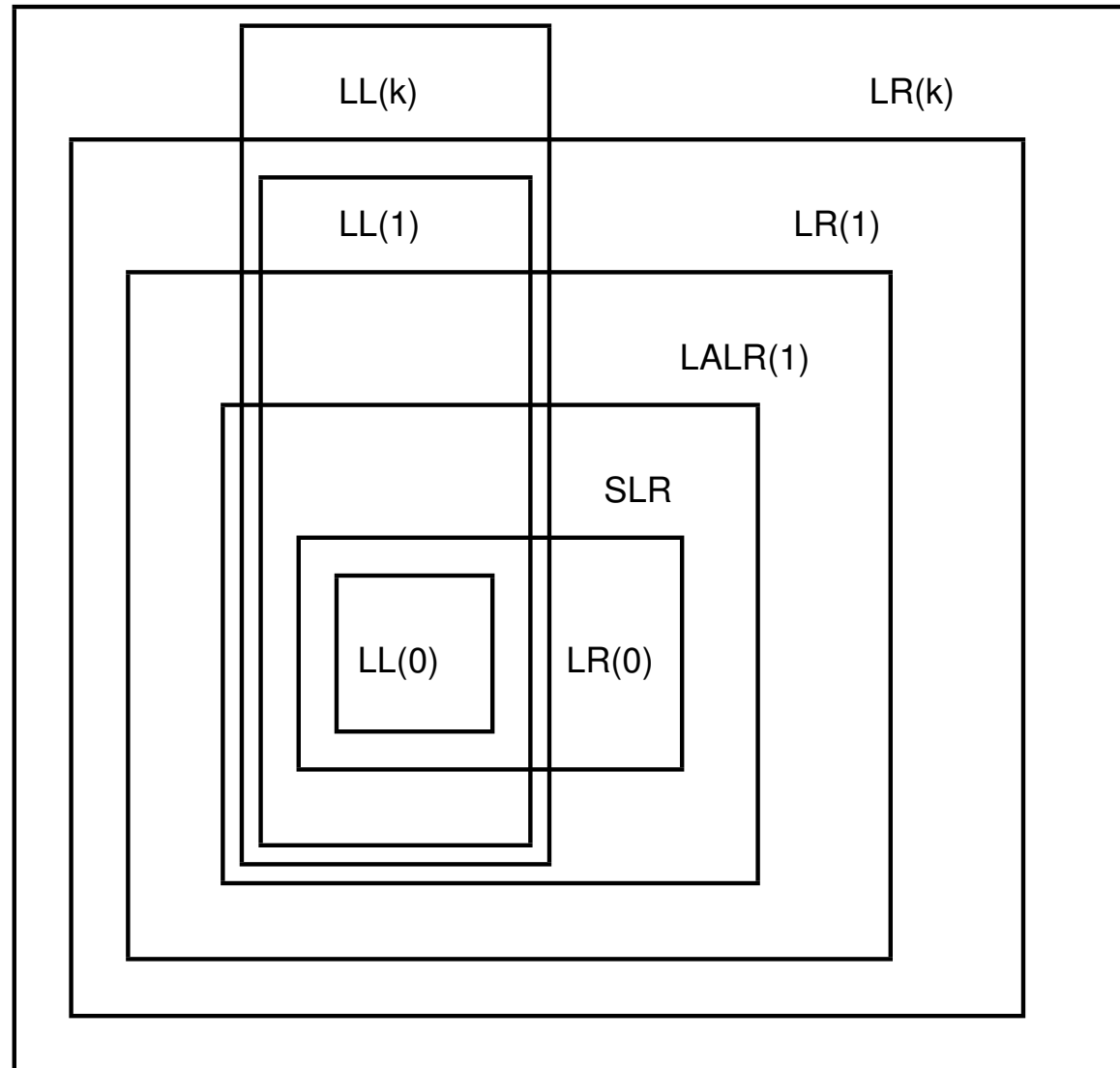
$$S_{\text{matched}} \rightarrow \text{"while" ident "do" } S_{\text{matched}}$$

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

$$\text{"else" } S_{\text{matched}}$$

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;
- Conflicts that can occur in LR parsers; 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;
- Shift-reduce conflicts in detail (a high-level understanding is enough for this class); and
- LALR parsers;

In past exams, questions on shift-reduce conflicts were bonus.