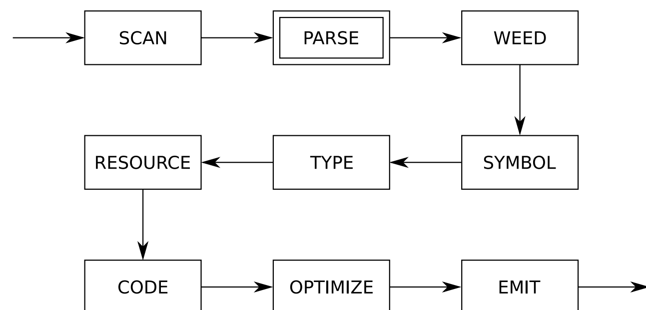
Parsing

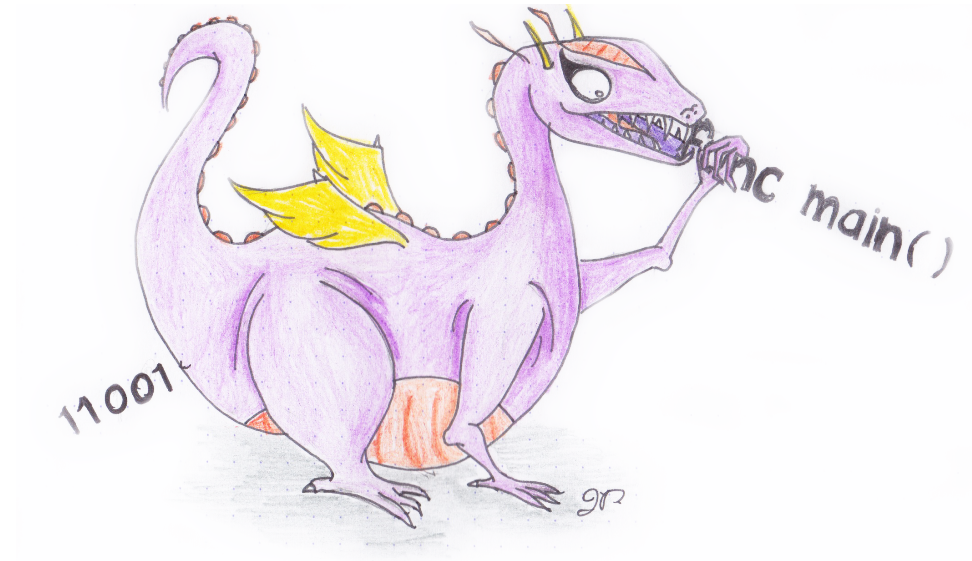
COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 13:30-14:30, MD 279





Announcements (Wednesday, January 11th)

Milestones:

- Continue forming your groups
- Learn `flex`, `bison`, `SableCC`
- Assignment 1 out today, due **Wednesday, January 25th 11:59PM** on myCourses

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/~cs520/2017/`)

- flex, bison, SableCC

Parsing:

- is the second phase of a compiler;
- takes a string of tokens generated by the scanner as input; and
- builis a *parse tree* according to some grammar.

Internally:

- it corresponds to a *deterministic push-down automaton*;
- plus some glue code to make it work;
- can be generated by `bison` (or `yacc`), CUP, ANTLR, SableCC, Beaver, JavaCC, . . .

A push-down automaton:

- 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
- are used to recognize context-free languages.

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

- V, a set of *variables* (or *non-terminals*)
- Σ, a set of *terminals* such that $V \cap \Sigma = \emptyset$
- R, a 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
- $S \in V$, the start variable

Context-free grammars:

- are stronger than regular expressions;
- are able to express recursively-defined constructs; and
- generate a context-free language.

For example: we cannot write a regular expression for any number of matched parentheses:

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

Using a CFG:

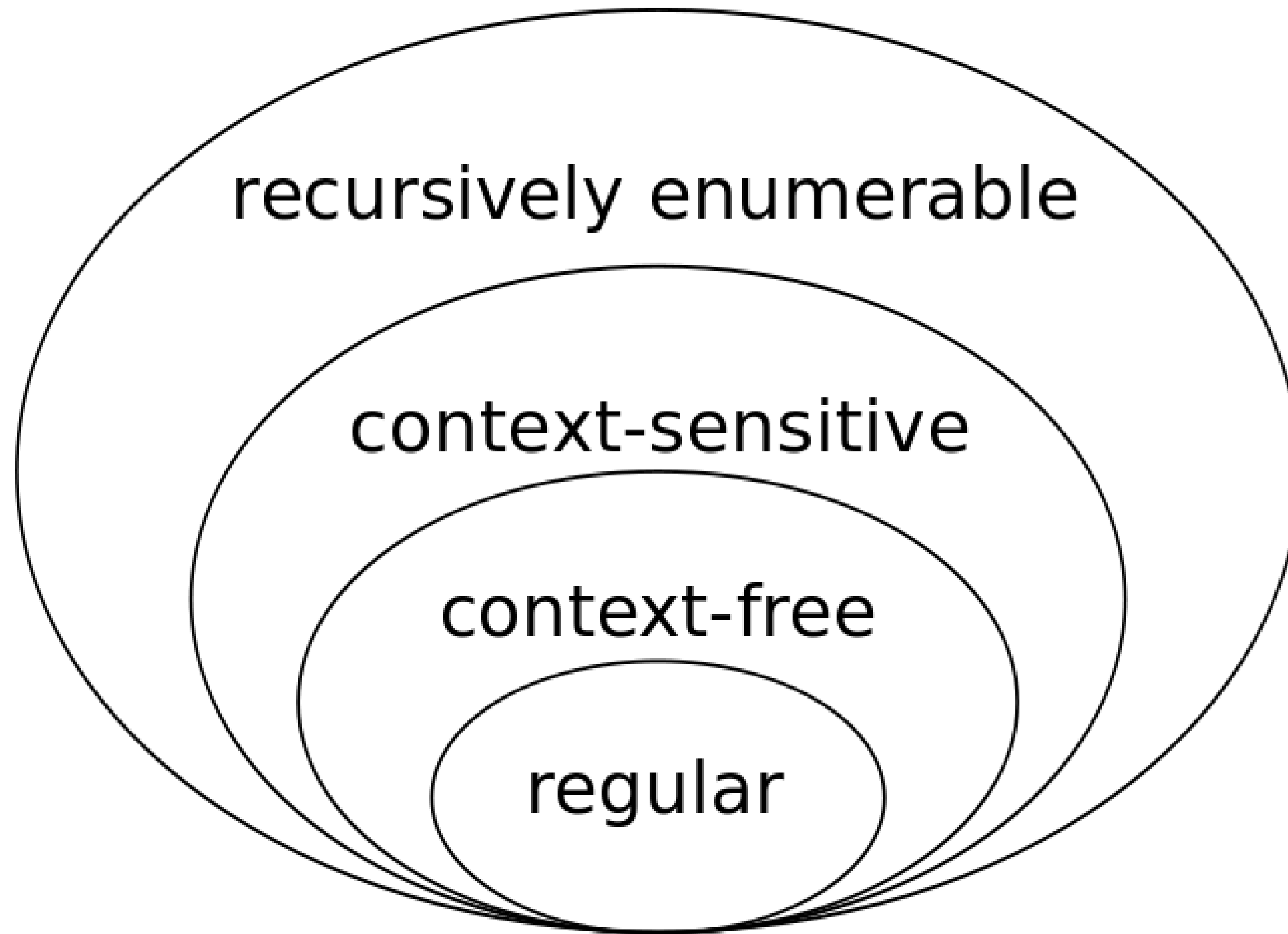
$$E \rightarrow (\ E\ ) \mid \epsilon$$

Notes on CFLs:

- 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;
- 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).

Chomsky Hierarchy:



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

Automated parser generators:

- use CFGs are input; and
- generate parsers using the machinery of a deterministic push-down automaton.

However, to be efficient:

- they limit the kind of CFGs that are allowed as input; and
- do not accept any valid context-free language.

An example:

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?

We can perform a *rightmost derivation* by repeatedly replacing variables with their RHS until only terminals remain:

$\underline{A}$

a $\underline{B}$

a b $\underline{B}$

a b b $\underline{B}$

a b b c

An example programming language:

CFG rules:

Prog $\longrightarrow$ Dcls Stmts

Dcls $\longrightarrow$ Dcl Dcls | ϵ

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

Stmts $\longrightarrow$ Stmt Stmts | ϵ

Stmt $\longrightarrow$ ident "=" Val

Val $\longrightarrow$ num | ident

Leftmost derivation:

$\underline{Prog}$

$\underline{Dcls}\ Stmts$

$\underline{Dcl}\ Dcls\ Stmts$

"int" 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

This derivation corresponds to the program:

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

Different grammar formalisms. First, consider BNF (Backus-Naur Form):

```
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)

Second, consider EBNF (Extended BNF):

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	

where '{' and '}' are like Kleene *'s in regular expressions.

Now, how to specify `stmt_list`**:** 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)

become:

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

EBNF also has an *optional*-construct. For example:

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

could be written as:

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

And similarly:

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

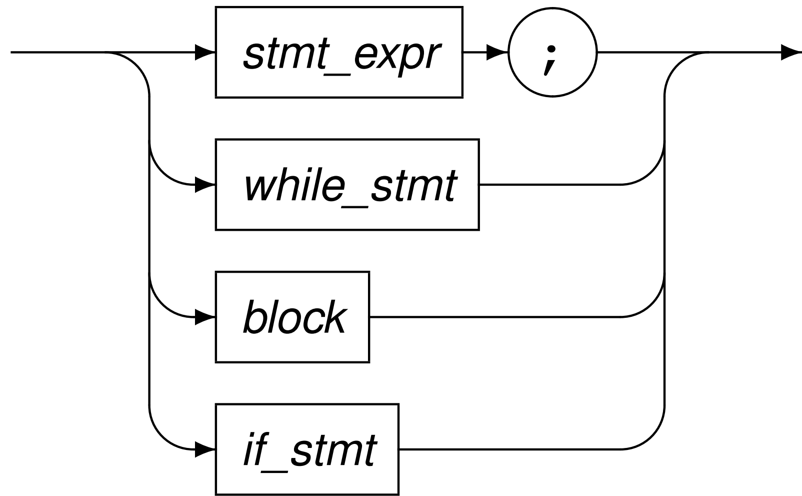
could be written as:

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

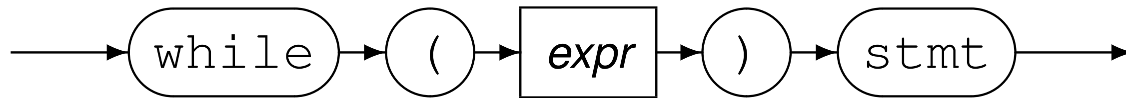
where '[' and ']' are like '?' in regular expressions.

Third, consider "railroad" syntax diagrams: (thanks rail.sty!)

stmt

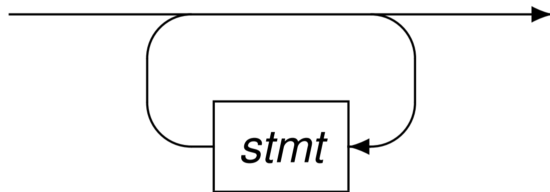


while_stmt

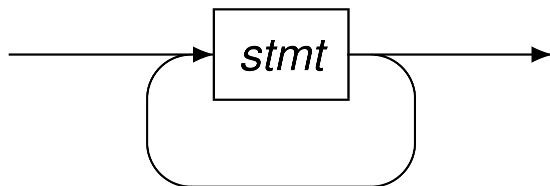


block

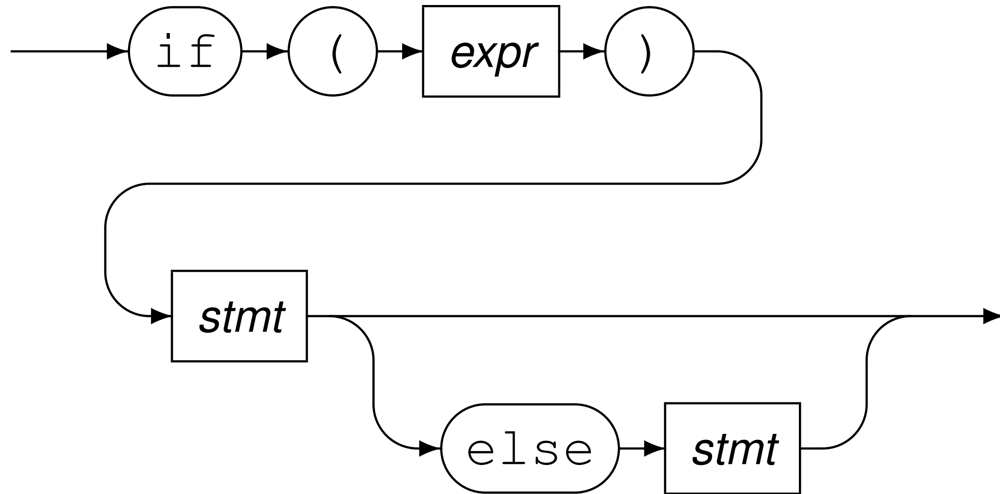
stmt_list (0 or more)



stmt_list (1 or more)



if_stmt



Derivations:

- consist of replacing variables with other variables and terminals according to the rules;
- i.e. for a rewrite rule A $\longrightarrow \gamma$, we replace A by γ.

Choosing the variable to rewrite:

- can be done as you wish; *but*
- in practice we either use *rightmost* or *leftmost* derivations;
- expanding the rightmost or leftmost variable respectively.
- *Note: this can lead to different parse trees!*

A parse tree:

- is a tree that represents the syntax structure of a string;
- is built from the rules given in a context-free grammar.

Nodes in the parse tree:

- internal (parent) nodes represent the LHS of a rewrite rule;
- child nodes represent the RHS of a rewrite rule;
- depend on the order of the derivation.

The *fringe* or leaves are the sentence you derived.

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

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)

This derivation corresponds to the program:

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

$S \rightarrow S$; S

$S \rightarrow$ id := E

$S \rightarrow$ print (L)

$E \rightarrow$ id

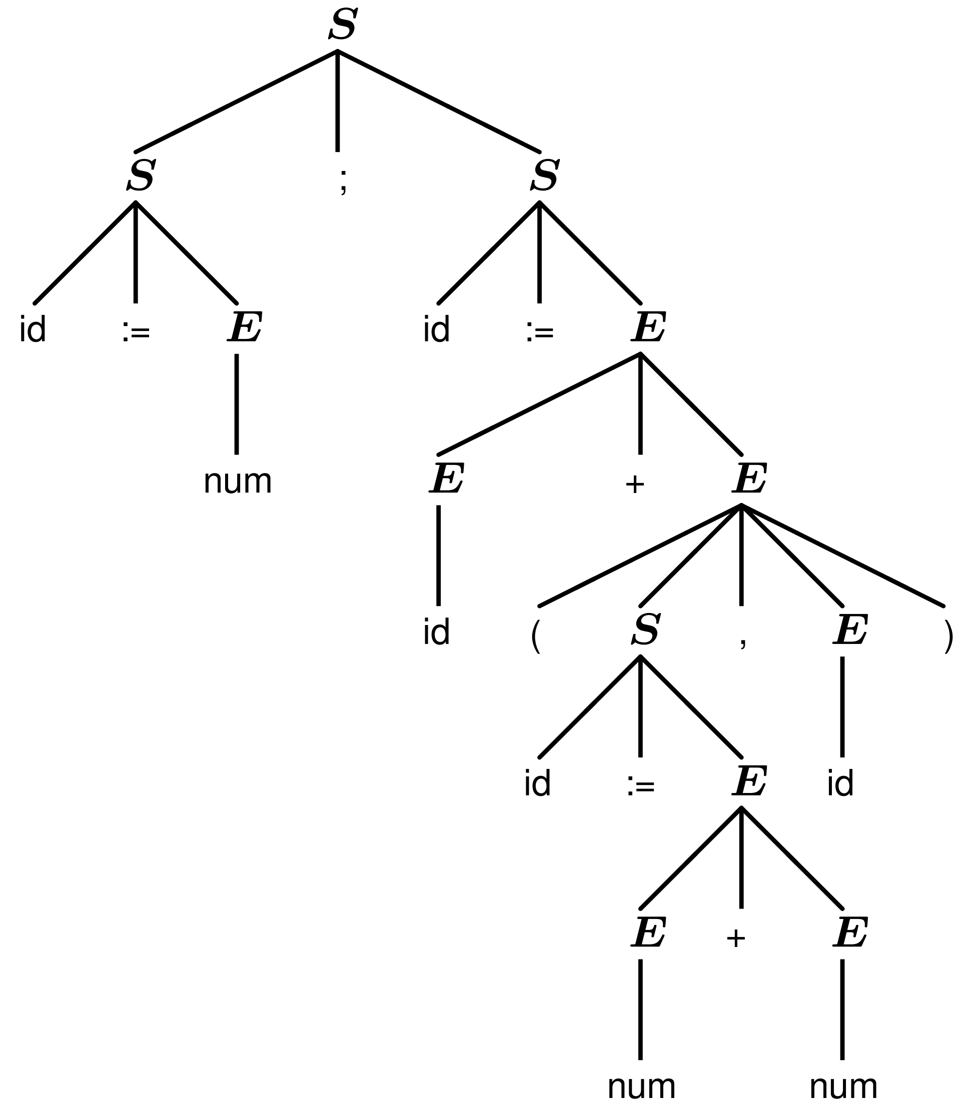
$E \rightarrow$ num

$E \rightarrow E$ + E

$E \rightarrow$ (S , E)

$L \rightarrow E$

$L \rightarrow L$, E

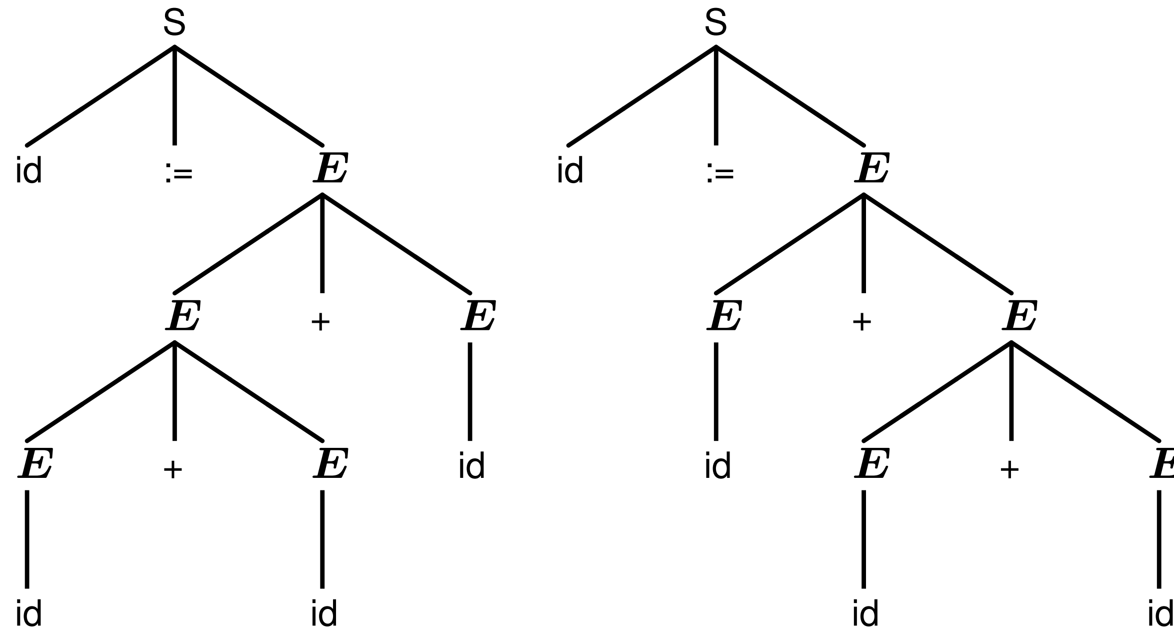


Derivation corresponds to the program:

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

A grammar is *ambiguous* if a sentence has different parse trees:

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



The above is harmless, but consider:

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

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

How do make grammars unambiguous?

- firstly, note that not all languages have an unambiguous grammar;
- however, deterministic push-down automata that are used by parsers, require an unambiguous grammar;
- in practice, we either rewrite the grammar to be unambiguous, or use precedence rules.

Rewriting an ambiguous grammar:

An ambiguous grammar:

$E \rightarrow$ id $E \rightarrow E \;/\; E$ $E \rightarrow (E)$

$E \rightarrow$ num $E \rightarrow E + E$

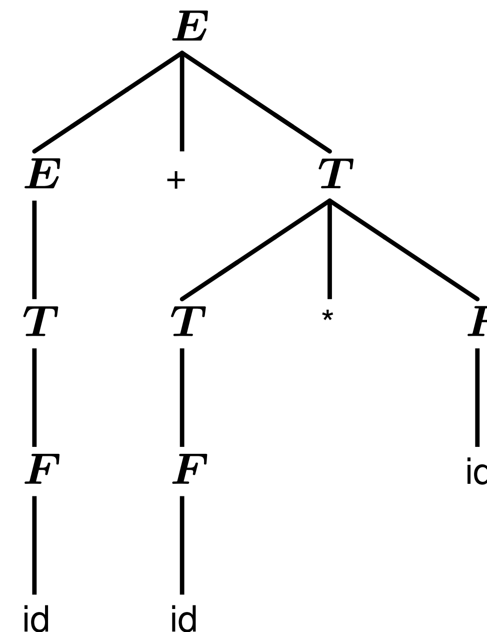
$E \rightarrow E * E$ $E \rightarrow E - E$

may be rewritten to become unambiguous:

$E \rightarrow E + T$ $T \rightarrow T * F$ $F \rightarrow$ id

$E \rightarrow E - T$ $T \rightarrow T \;/\; F$ $F \rightarrow$ num

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



Recall that parsers:

- take a string of tokens generated by the scanner as input; and
- buils a *parse tree* according to some grammar.
- *note: this corresponds to checking a string is in a language.*
- there are fundamentally two kinds of parsers:
  1. Top-down, *predictive* or *recursive descent* parsers. Used in all languages designed by Wirth, e.g. Pascal, Modula, and Oberon.
  2. Bottom-up parsers.

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:** look at beginning of input (up to k characters) and unambiguously expand leftmost non-terminal.

A top-down parser:

- begins with the start symbol (root); and
- grows the parse tree using the defined grammar.
- this is *predictive*: the parser must determine (given some input) which rule to apply next.

Recall the definition of LL(*k*):

- *Left-to-right parse*;
- *Leftmost-derivation*; and
- *k symbol lookahead*.

An example LL(1) parsing:

Given the CFG:

Prog $\longrightarrow$ Dcls Stmts

Dcls $\longrightarrow$ Dcl Dcls | ϵ

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

Stmts $\longrightarrow$ Stmt Stmts | ϵ

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

Val $\longrightarrow$ num | ident

Parse the program:

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

The token string generated by a scanner is:

```
tINT
tIDENTIFIER: a
tFLOAT
tIDENTIFIER: b
tIDENTIFIER: a
tASSIGN
tIDENTIFIER: b
```

Top-down parsers:

- are usually implemented as *recursive descent*;
- i.e. a set of mutually recursive functions that:
  - predict which rule to apply; and
  - apply the rules/productions:
    - consume/match terminals; and
    - recursively call functions for other non-terminals.
- can also be implemented as a table driven approache (textbook Chapter 5.4).

A recursive descent parser:

- has 1 function for each non-terminal (variable);
- each non-terminal has a *predict set* for each of its rules;
- 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;
  - none of the predict sets: then there is a syntax error.

For a subset of the example CFG:

Prog $\longrightarrow$ Dcls Stmts

Dcls $\longrightarrow$ Dcl Dcls | ϵ

Dcl $\longrightarrow$ "int" ident | "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()
        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
```

Limitations of this approach (common prefixes):

Consider the following productions, defining an If-Else-End construct:

IfStmt $\longrightarrow$ tIF Stmts tEND | tIF 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 expect the token tIF).

To get around this problem, we *factor* the grammar:

IfStmt $\longrightarrow$ tIF Stmts IfEnd

IfEnd $\longrightarrow$ tEND | tELSE Stmts tEND

Now, each production for IfEnd has different predict token (the predict sets have null intersection)

Limitations of this approach (left recursion):

Left recursion also causes difficulties with LL(k) parsers. Consider the following production:

$$A \longrightarrow A\ \beta$$

Assume we can come up with a predict set consisting of token, tTOKEN. Then applying this rule gives us:

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

This continues on forever. *note there are other ways to think of this*

The dangling else problem:

IfStmt $\longrightarrow$ 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?

To resolve this ambiguity we associate the else with the *nearest unmatched* if-statement. Note that the grammar we come up with is still not LL(k) - see textbook Chapter 5.6 for more details.

Announcements (Friday, January 13th)

Milestones:

- Continue forming your groups
- Learn `flex`, `bison`, `SableCC`
- Assignment 1 due **Wednesday, January 25th 11:59PM** on myCourses
- Add/drop deadline, **Tuesday, January 17th**

Assignment 1:

- Due **Wednesday, January 25th 11:59PM** on myCourses
- Questions about the assignment?
- Questions about the language?

Recall:

A *parser* transforms a string of tokens into a parse tree, according to some grammar:

- it corresponds to a *deterministic push-down automaton*;
- plus some glue code to make it work;
- can be generated by `bison` (or `yacc`), CUP, ANTLR, SableCC, Beaver, JavaCC, ...

(Review) 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:** look at beginning of input (up to k characters) and unambiguously expand leftmost non-terminal.

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**: look for a sequence matching RHS and reduce to LHS. Postpone any decision until entire RHS is seen, plus k tokens lookahead.

Bottom-up parsers:

- build parse trees from the leaves to the root;
- performs a rightmost derivation in reverse; and
- uses 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.

The *shift-reduce* bottom-up parsing technique

1) Extend the grammar with an end-of-file \$, introduce fresh start symbol S':

$S' \rightarrow S$\$

$S \rightarrow S$; 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)	

2) Choose between the following actions:

- shift:

  move first input token to top of stack

- reduce:

  replace α on top of stack by X

  for some rule $X \rightarrow \alpha$

- accept:

  when S' is on the stack

An example:

Stack	Input	Action
	`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

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

Recall the previous rightmost derivation of this 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*.

Internally, shift-reduce parsers:

- are implemented as a stack of states;
- states represent which tokens have been processed (are on the left side), *without* having to scan the contents;
- shift/reduce according to the current state, and the next *k* unprocessed tokens.
- *Note how this resembles a DFA.*

We can implement this logic using a standard parser driver:

```
while not accepted do
    action = LookupAction(currentState, nextTokens)
    if action == shift<nextState>
        push(nextState)
    else if action == reduce<A->stuff>
        pop(|stuff|)
        push(NextState(currentState, A))
    else
        error()
done
```

Back to our example:

- each rule is given a number:

$_0\ S' \rightarrow S\$$	$_5\ E \rightarrow$ num
$_1\ S \rightarrow S$; S	$_6\ E \rightarrow E$ + E
$_2\ S \rightarrow$ id := E	$_7\ E \rightarrow$ (S , E)
$_3\ S \rightarrow$ print (L)	$_8\ L \rightarrow E$
$_4\ E \rightarrow$ id	$_9\ L \rightarrow L$, E

- start with initial state (s1) on the stack;
- we 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 $X \rightarrow \alpha$; pop $|\alpha|$ times; lookup (stack top, X) in table
  - goto(n): push state n
  - accept: report success

DFA	terminals										non-terminals		
state	id	num	print	;	,	+	:=	(	)	$	S	E	L
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 omitted.

Stack / Action	Input
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	
$s_1\ s_4\ s_6$ ~~s_{10}~~	$
lookup(s_6,E) = goto(11)	
$s_1\ s_4\ s_6\ s_{11}$	$
reduce(2): $S \longrightarrow$ id := E	
s_1 ~~s_4~~ ~~s_6~~ ~~s_{11}~~	$
lookup(s_1,S) = goto(2)	
$s_1\ s_2$	$
accept	

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 (shift/reduce, reduce/reduce) arise, then we are happy; otherwise, fix grammar.

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

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

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

The sequence α is on top of the stack, and the head of the input is derivable from $\beta\gamma$x. There are two cases for β, terminal or non-terminal.

We first compute a set of LR(1) states from our grammar, and then use them to build a parse table. There are four kinds of entry to make:

1. goto: when β is non-terminal
2. shift: when β is terminal
3. reduce: when β is empty (the next state is the number of the production used)
4. accept: when we have A $\longrightarrow$ B . $

Follow construction on the tiny grammar:

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

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

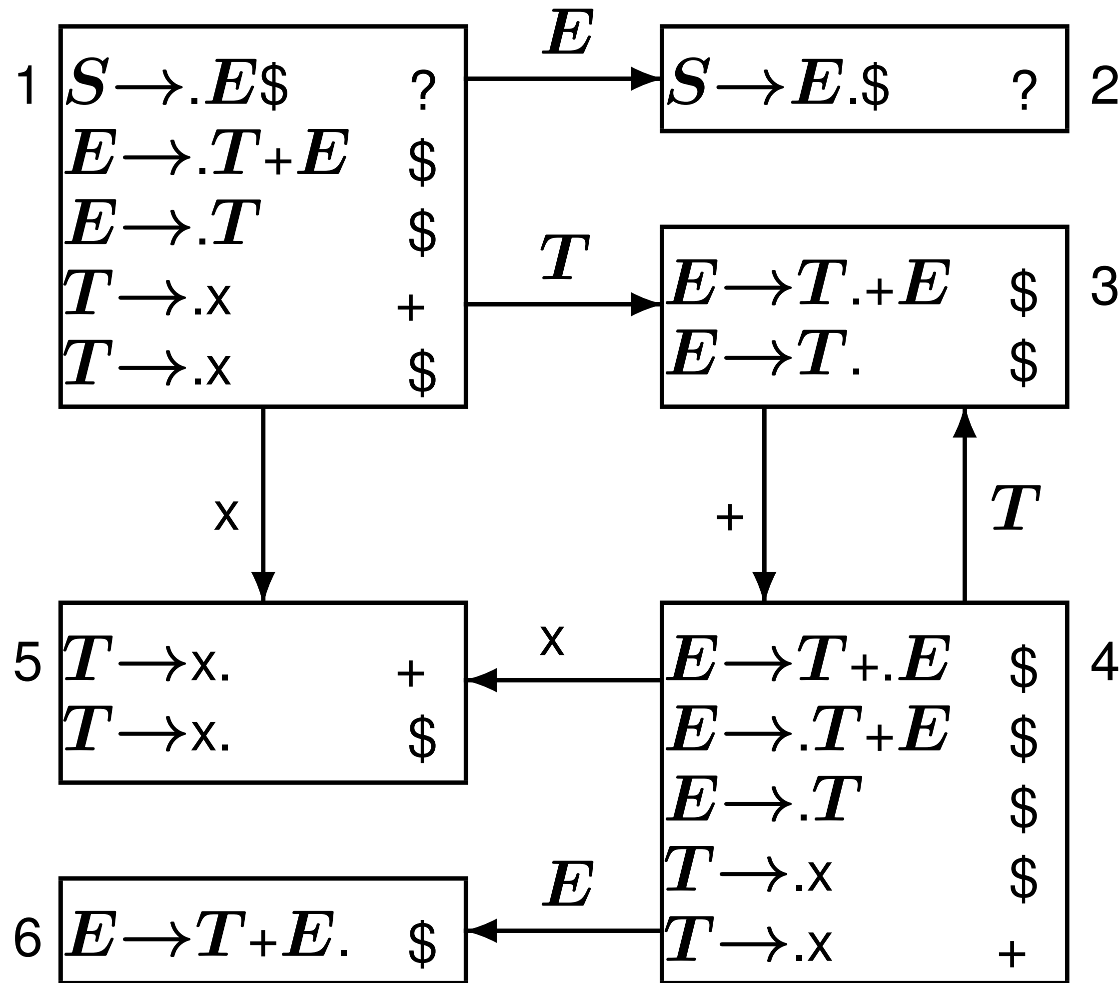
Constructing the LR(1) NFA:

- start with state | $S \rightarrow . E\$$? |
- state | $A \rightarrow \alpha . B \beta$ l | has:
  - ϵ-successor | $B \rightarrow . \gamma$ x |, if:
    - exists rule $B \rightarrow \gamma$, and
    - x $\in$ lookahead(β)
  - B-successor | $A \rightarrow \alpha B . \beta$ l |
- state | $A \rightarrow \alpha$. x β l | has:

  x-successor | $A \rightarrow \alpha$ x . β l |

Constructing the LR(1) DFA:

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		

Conflicts

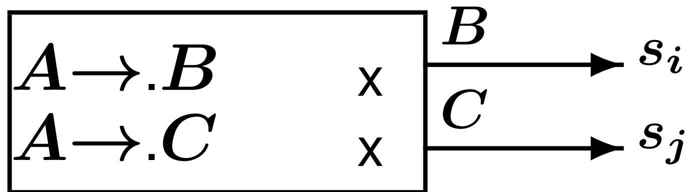
Items	Lookahead	
$A \rightarrow .B$	x	no conflict (lookahead decides)
$A \rightarrow C.$	y	

Items	Lookahead	
$A \rightarrow .B$	x	shift/reduce conflict
$A \rightarrow C.$	x	

Items	Lookahead	
$A \rightarrow .$x	y	shift/reduce conflict
$A \rightarrow C.$	x	

Items	Lookahead	
$A \rightarrow B.$	x	reduce/reduce conflict
$A \rightarrow C.$	x	

What about shift/shift conflicts?

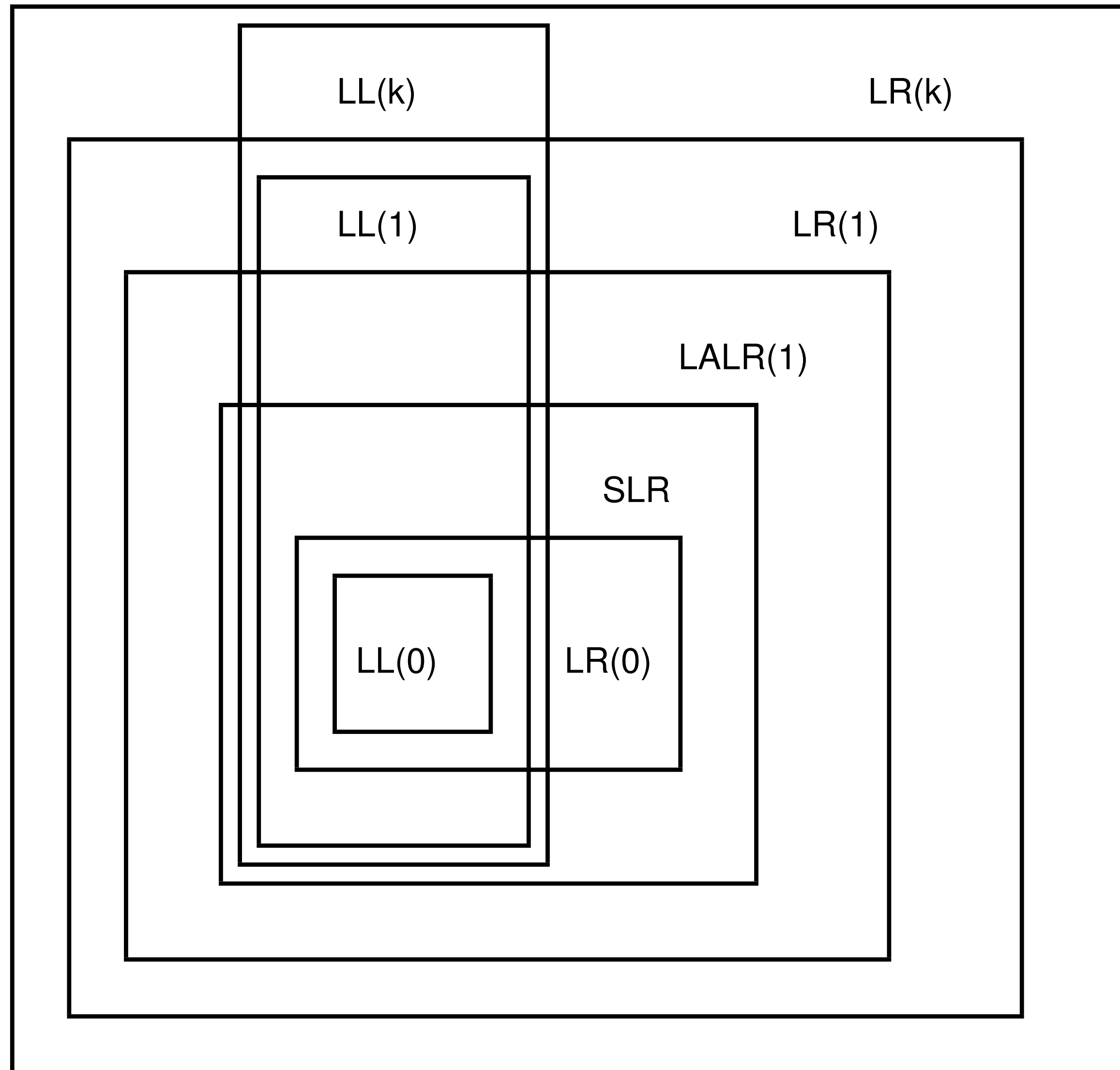


$\Rightarrow$ by construction of the DFA

we have $s_i = s_j$

LR(1) tables may become very large.

Parser generators use LALR(1), which merges states that are identical except for lookaheads.



LL(k)
LR(k)
LL(1)
LR(1)
LALR(1)
SLR
LL(0)
LR(0)

Takeaways:

You will not be asked to build a parser DFA/NFA/Table on the exams, but you should understand:

- what it means to shift and reduce;
- conflicts that can occur when generating a parser;
- the general idea of how the table or DFA is used during a parse.

Announcements (Monday, January 16th)

Milestones:

- Group formation should be complete this week - a signup sheet will be distributed after Add/Drop
- Assignment 1 due **Wednesday, January 25th 11:59PM** on myCourses
- Add/drop deadline, tomorrow, **Tuesday, January 17th**

Assignment 1:

- Questions?
- No TA office hours tomorrow

Reference compiler (minilang):

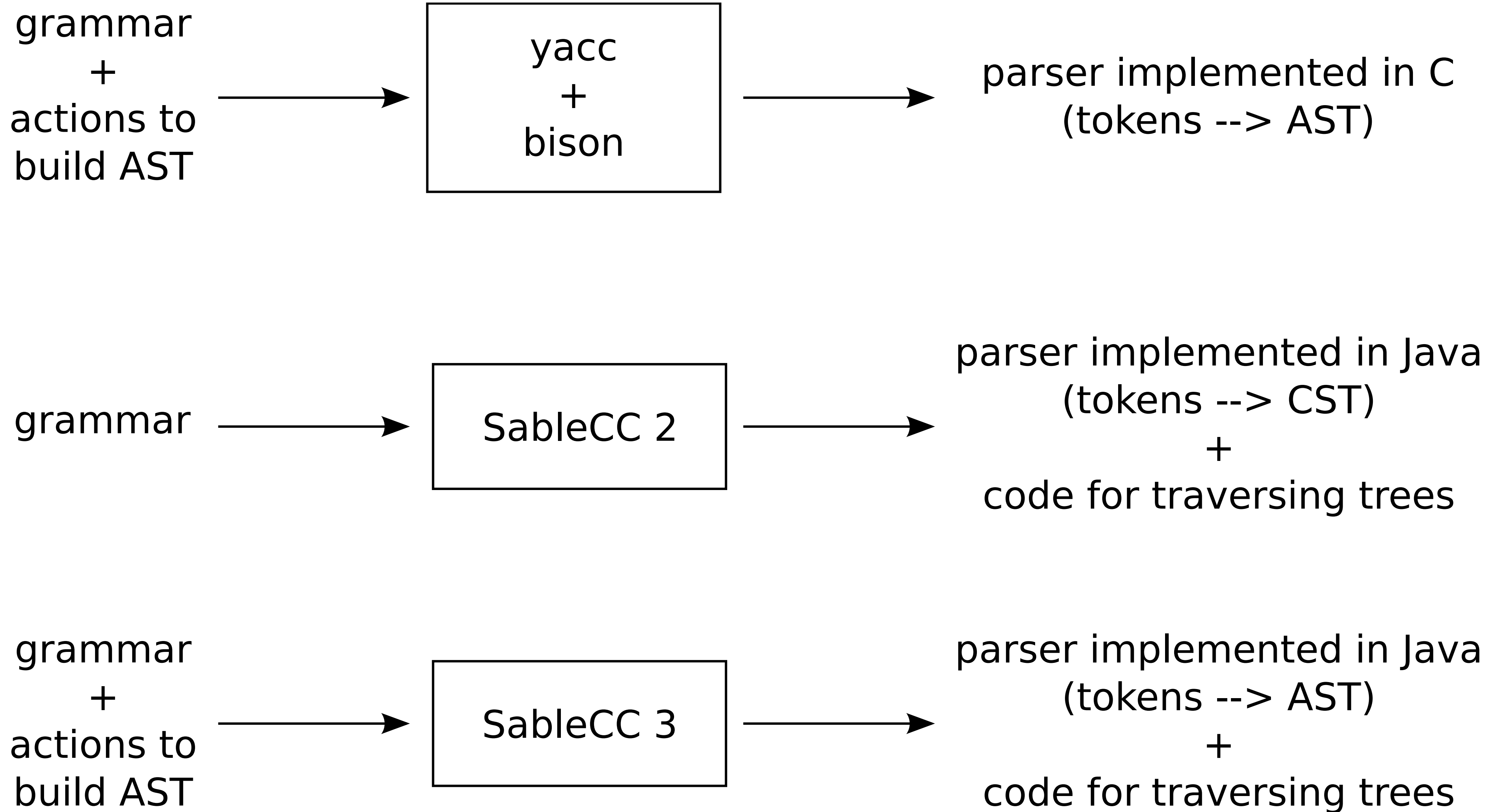
- ssh <socs_username>@teaching.cs.mcgill.ca
- ~cs520/minilang/minic {keyword} < {file}
- if you find errors in the reference compiler, up to 5 bonus points on the assignment

Keywords for the first assignment:

- scan: run scanner only, VALID/INVALID
- tokens: produce the list of tokens for the program
- parse: run scanner+parser, VALID/INVALID

Run script should be out soon

LALR Parser Tools



`bison` **(**`yacc`**) is a parser generator:**

- it inputs a grammar;
- it computes an LALR(1) parser table;
- it reports conflicts;
- it resolves conflicts using defaults (!); and
- it creates a C program.

Nobody writes (simple) parsers by hand anymore.

The grammar:

1 $E \rightarrow \text{id}$

2 $E \rightarrow \text{num}$

3 $E \rightarrow E * E$

4 $E \rightarrow E \, / \, E$

5 $E \rightarrow E + E$

6 $E \rightarrow E - E$

7 $E \rightarrow (E)$

is expressed in `bison` as:

```
%{
    /* C declarations */
%}
/* Bison declarations; tokens come from lexer (scanner) */
%token tIDENTIFIER tINTCONST
%start exp
/* Grammar rules after the first %% */
%%
exp : tIDENTIFIER
    | tINTCONST
    | exp '*' exp
    | exp '/' exp
    | exp '+' exp
    | exp '-' exp
    | '(' exp ')'
;
%% /* User C code after the second %% */
```

The grammar is ambiguous:

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

$ cat exp.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.

[...]
```

With more details about each state

```
state 11

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)
```

Rewrite the grammar to force reductions:

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

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

$$E \rightarrow T \qquad T \rightarrow F \qquad F \rightarrow (\, E \,)$$

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

Or use precedence directives:

```
%token tIDENTIFIER tINTCONST
%start exp
%left '+' '-'      /* left-associative, lower precedence */
%left '*' '/'      /* left-associative, higher precedence */

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

Which resolve shift/reduce conflicts:

```
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.

The precedence directives are:

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

When constructing a parse table, an action is chosen based on the precedence of the last symbol on the right-hand side of the rule.

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

If precedences are equal, then:

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

This usually ends up working.

Using –report we can see the full error:

```
state 0
       tIDENTIFIER shift, and go to state 1
       tINTCONST   shift, and go to state 2
       '('         shift, and go to state 3
       exp         go to state 4
state 1
       exp  ->  tIDENTIFIER .   (rule 1)
       $default    reduce using rule 1 (exp)
state 2
       exp  ->  tINTCONST .   (rule 2)
       $default    reduce using rule 2 (exp)
...
state 14
       exp  ->  exp . '*' exp   (rule 3)
       exp  ->  exp . '/' exp   (rule 4)
       exp  ->  exp '/' exp .   (rule 4)
       exp  ->  exp . '+' exp   (rule 5)
       exp  ->  exp . '-' exp   (rule 6)
       $default    reduce using rule 4 (exp)
state 15
       $           go to state 16
state 16
       $default    accept
```

```
$ cat exp.y

%{
    #include <stdio.h>   /* for printf */
    extern char *yytext; /* string from scanner */
    void yyerror() { printf ("syntax error before %s\n", yytext); }
%}
%union {
    int intconst;
    char *stringconst;
}
%token <intconst> tINTCONST
%token <stringconst> tIDENTIFIER
%start exp
%left '+' '-'
%left '*' '/'
%%
exp : tIDENTIFIER { printf ("load %s\n", $1); }
    | tINTCONST   { 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 ')' {}
;
%%
```

```
$ cat exp.l
%{
    #include "y.tab.h"  /* for exp.y types */
    #include <string.h> /* for strlen */
    #include <stdlib.h> /* for malloc and atoi */
%}
%%
[ \t\n]+  /* ignore */;
"*"       return '*';
"/"       return '/';
"+"       return '+';
"-"       return '-';
"("       return '(';
")"       return ')';
0|([1-9][0-9]*) {
    yylval.intconst = atoi (yytext);
    return tINTCONST;
}
[a-zA-Z_][a-zA-Z0-9_]* {
    yylval.stringconst =
    (char *) malloc (strlen (yytext) + 1);
    sprintf (yylval.stringconst, "%s", yytext);
    return tIDENTIFIER;
}
.         /* ignore */
%%
```

Invoking the scanner and parser requires calling `yyparse`:

```
$ cat main.c
void yyparse();
int main (void)
{
    yyparse ();
}
```

Using `flex`/`bison` to create a parser is simple:

```
$ flex exp.l
$ bison --yacc --defines exp.y # note compatability options
$ gcc lex.yy.c y.tab.c y.tab.h main.c -o exp -lfl
```

An example:

When input `a*(b-17) + 5/c`:

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

our `exp` parser outputs the correct order of operations:

```
load a
load b
push 17
minus
mult
push 5
load c
div
plus
```

You should confirm this for yourself!

Error recovery:

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

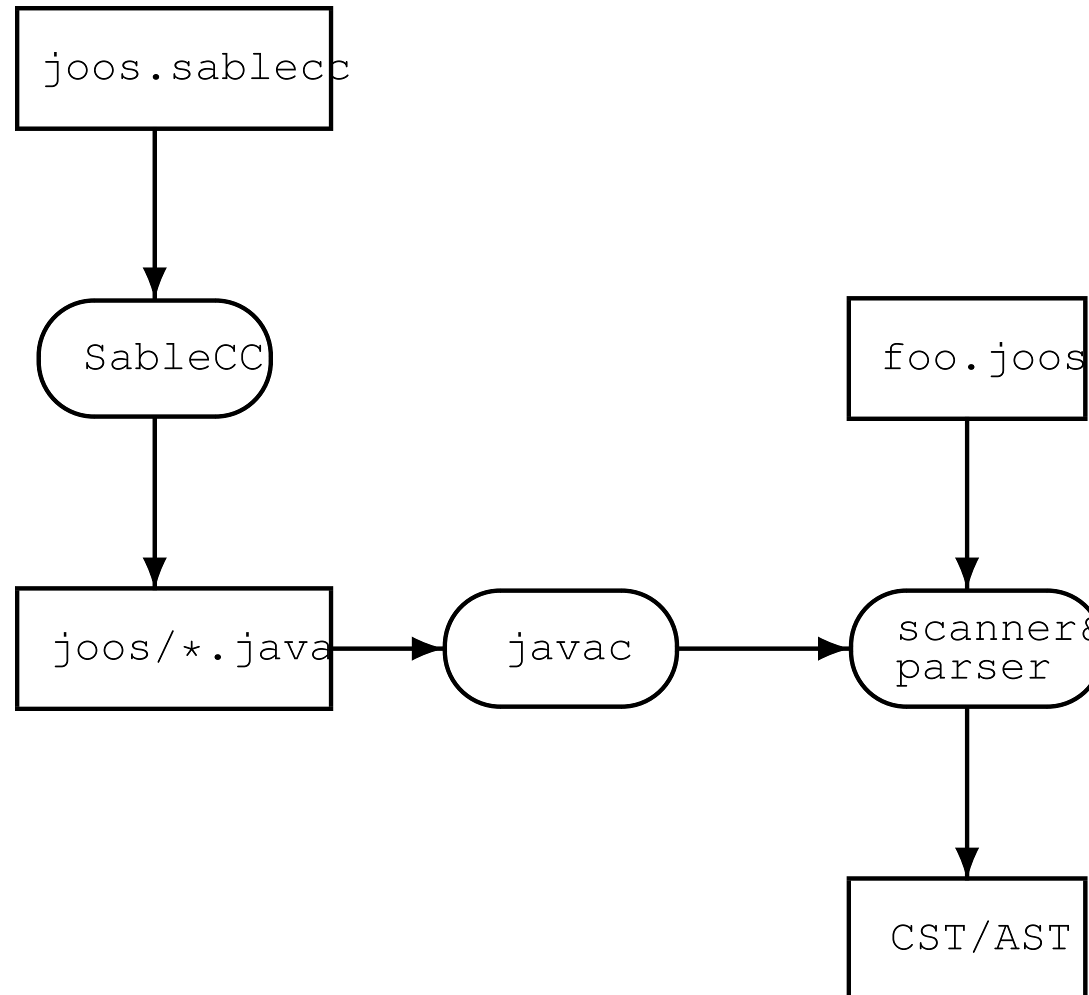
We may ask it to recover from the error:

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

and on input `a@(b-17) ++ 5/c` 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
```

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 for it.



The SableCC 2 grammar for our Tiny language:

```
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;
```

```
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;
```

Version 2 produces parse trees, a.k.a. concrete syntax trees (CSTs).

The SableCC 3 grammar for our Tiny language:

```
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)};
```

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;
```

Version 3 generates abstract syntax trees (ASTs).

A bit more on SableCC and ambiguities

The next slides are from "Modern Compiler Implementation in Java", by Appel and Palsberg.

GRAMMAR 3.30

1. $P \rightarrow L$
2. $S \rightarrow$ id := id
3. $S \rightarrow$ while id do S
4. $S \rightarrow$ begin L end
5. $S \rightarrow$ if id then S
6. $S \rightarrow$ if id then S else S
7. $L \rightarrow S$
8. $L \rightarrow L$; S

First part of SableCC specfication (scanner)

GRAMMAR 3.32: SableCC version of Grammar 3.30.

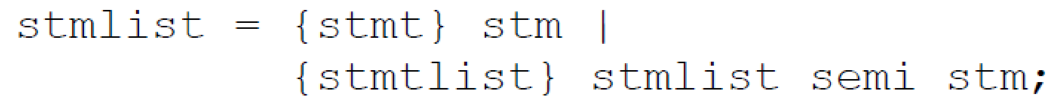
```
Tokens
    while = 'while';
    begin = 'begin';
    end = 'end';
    do = 'do';
    if = 'if';
    then = 'then';
    else = 'else';
    semi = ';';
    assign = '=';
    whitespace = (' ' | '\t' | '\n')+;
    id = ['a'..'z'](['a'..'z'] | ['0'..'9'])*;
Ignored Tokens
    whitespace;
```

Second part of SableCC specfication (parser)

```
Productions
    prog = stmlist;

    stm =  {assign} [left]:id assign [right]:id |
           {while} while id do stm |
           {begin} begin stmlist end |
           {if_then} if id then stm |
           {if_then_else} if id then [true_stm]:stm else [false_stm]:stm;

    stmlist = {stmt} stm |
              {stmtlist} stmlist semi stm;
```



Shift reduce confict because of ”dangling else problem"

```
shift/reduce conflict in state [stack: TIf TId TThen PStm *] on TElse in {
        [ PStm = TIf TId TThen PStm * TElse PStm ] (shift),
        [ PStm = TIf TId TThen PStm * ] followed by TElse (reduce)
}
```

Figure 3.33: SableCC shift-reduce error message for Grammar 3.32.

GRAMMAR 3.34: SableCC productions of Grammar 3.32 with conflicts resolved.

```
Productions
    prog = stmlist;

    stm = {stm_without_trailing_substm}
               stm_without_trailing_substm |
          {while} while id do stm |
          {if_then} if id then stm |
          {if_then_else} if id then stm_no_short_if
                         else [false_stm]:stm;

    stm_no_short_if = {stm_without_trailing_substm}
                         stm_without_trailing_substm |
                        {while_no_short_if}
                         while id do stm_no_short_if |
                     {if_then_else_no_short_if}
                        if id then [true_stm]:stm_no_short_if
                              else [fals_stm]:stm_no_short_if;

    stm_without_trailing_substm = {assign} [left]:id assign [right]:id |
                                     {begin} begin stmlist end ;
    stmlist = {stmt} stm | {stmtlist} stmlist semi stm;
```

Shortcut for giving precedence to unary minus in bison/yacc

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
```