Syntactic Analysis

Context Free Grammars

- Regular Expressions are unable to specify nested constructs
- Consider the following:

  expr → **id** | **number** | - *expr* | (*expr*)

  | *expr op expr*

  op → + | - | * | /

- **Tokens in bold**
- *Non-terminals in italic*
- | means "or"
- → means "can be replaced by"

- Need the ability to represent recursion, something in terms of itself
- For instance, cannot describe matching parenthesis with RE alone

Context Free Grammars (CFG)

```
expr  → id
      | number
      | - expr
      | ( expr )
      | expr op expr
op    → +
      | -
      | *
      | /
```

- Notation known as Backus-Naur Form (BNF)
- Owed to John Backus and Peter Naur who designed it for Algol-60

- CFG or grammar for short
- "Context Free" means that rules are applicable independently of the context or surroundings
- Each rule in a CFG is a *production*
- Symbols on the left-hand side are *variables* or *non-terminals*
- A variable can have any number of productions
- *Tokens* (symbols in bold) are also known as *terminals* (because they don't have a right-hand sides or productions)
- Tokens cannot appear on the left-hand side of productions
- Some non-terminal is chosen as the *start symbol*

Context Free Grammars (CFG)

```
expr  → id
      →number
expr  → - expr
      | ( expr )
      | expr op expr
op    → +
      | -
      | *
      | /
```

- If I forget to mark some symbol in bold or italic, the role of the symbol can be easily determined from its context
- It’s totally fine to write productions either with → or with |

Context-Free Grammars

In summary, a CFG consists of:

- a set of terminals T
- a set of non-terminals N
- a non-terminal S identified as the start symbol
- a set of productions

Scanning and Parsing

- Recall that tokens such as IDENTIFIER and NUMBER actually represent sets of strings acceptable by a language
- The parser, however, does not distinguish between “1.5”, “1” or “10000.0000”.
- The actual values (as number, strings or names) are stored in the symbol table along the parsing process

Derivation and Parse Trees

A CFG allows to generate syntactically valid string of terminals, i.e. a valid program:

a. Begin with start symbol
b. Choose a production with the start symbol on the left-hand side
c. Replace start symbol with right-hand side of production
d. Choose a non-terminal A in resulting string
e. Choose a production P with A on the left-hand side, and replace A with the right-hand side of P

Context-Free Grammars

Consider the previous grammar for arithmetic expressions:

$$
\begin{aligned}
expr &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\
&\quad\ \mid expr\ op\ expr \\
op &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

How could we produce the string “slope * x + intercept” ?

Context-Free Grammars

$$
\begin{aligned}
expr &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\
&\quad\ \mid expr\ op\ expr \\
op &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

slope * x + intercept

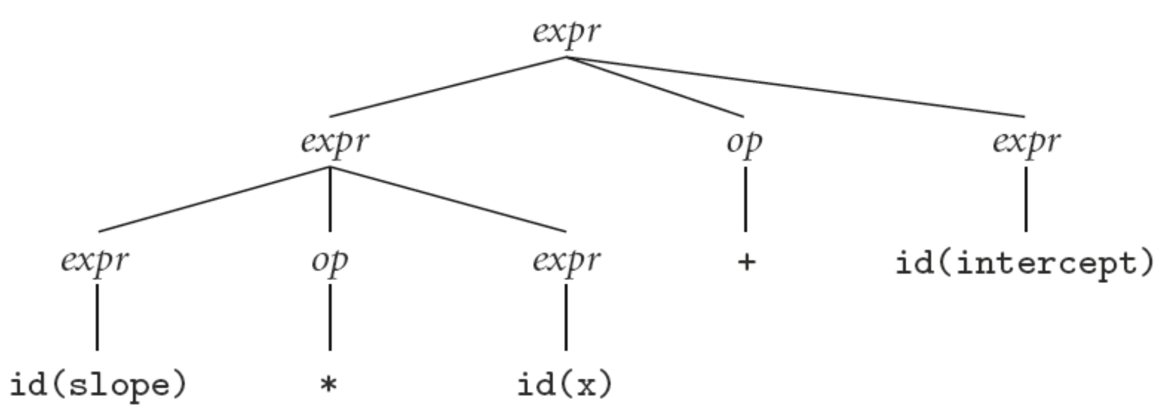
$$
\begin{aligned}
expr &\Longrightarrow expr\ op\ \underline{expr} \\
&\Longrightarrow expr\ \underline{op}\ \texttt{id} \\
&\Longrightarrow \underline{expr}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow expr\ op\ \underline{expr}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow expr\ \underline{op}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \underline{expr}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \underset{\texttt{(slope)}}{\texttt{id}}\ \texttt{*}\ \underset{\texttt{(x)}}{\texttt{id}}\ \texttt{+}\ \underset{\texttt{(intercept)}}{\texttt{id}}
\end{aligned}
$$

Context-Free Grammars

$$
\begin{aligned}
expr &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\
&\quad \mid expr\ op\ expr \\
op &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

slope * x + intercept

$$
\begin{aligned}
expr &\Longrightarrow expr\ op\ \underline{expr} \\
&\Longrightarrow expr\ \underline{op}\ \texttt{id} \\
&\Longrightarrow \underline{expr}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow expr\ op\ \underline{expr}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow expr\ \underline{op}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \underline{expr}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \underset{\texttt{(slope)}}{\texttt{id}}\ \texttt{*}\ \underset{\texttt{(x)}}{\texttt{id}}\ \texttt{+}\ \underset{\texttt{(intercept)}}{\texttt{id}}
\end{aligned}
$$



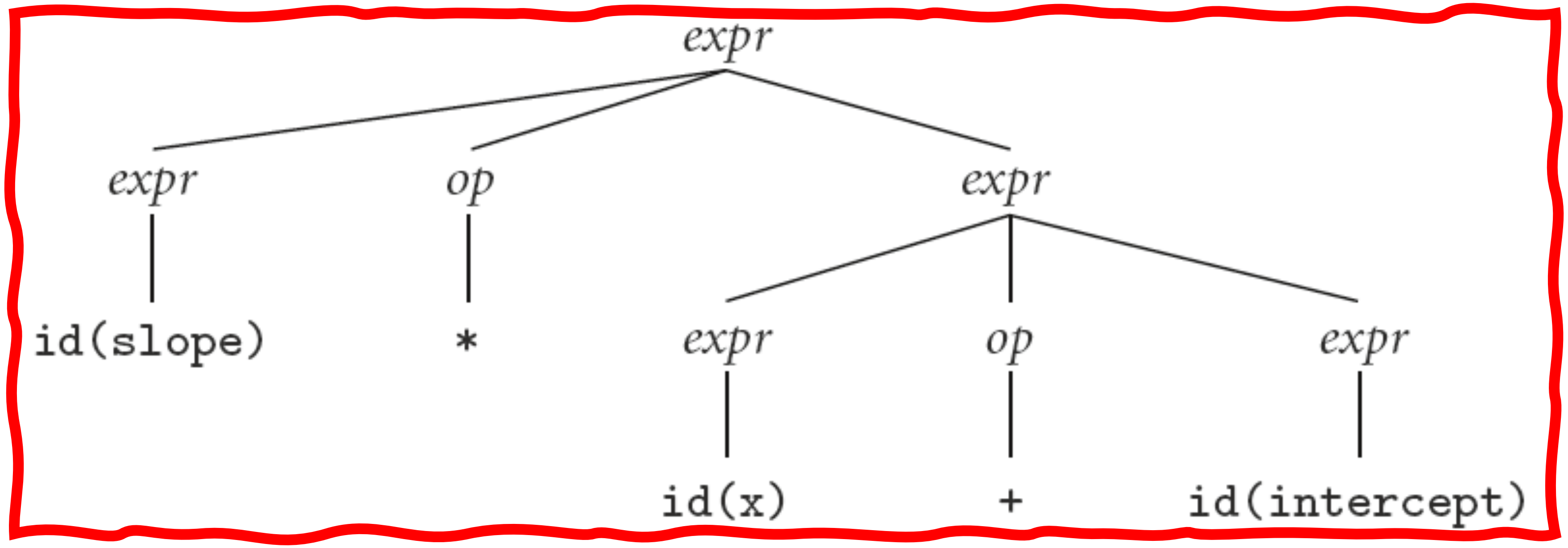
Sentential forms

A parse tree

Context-Free Grammars

$$
\begin{aligned}
expr &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\
&\quad\ \mid expr\ op\ expr \\
op &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

slope * x + intercept

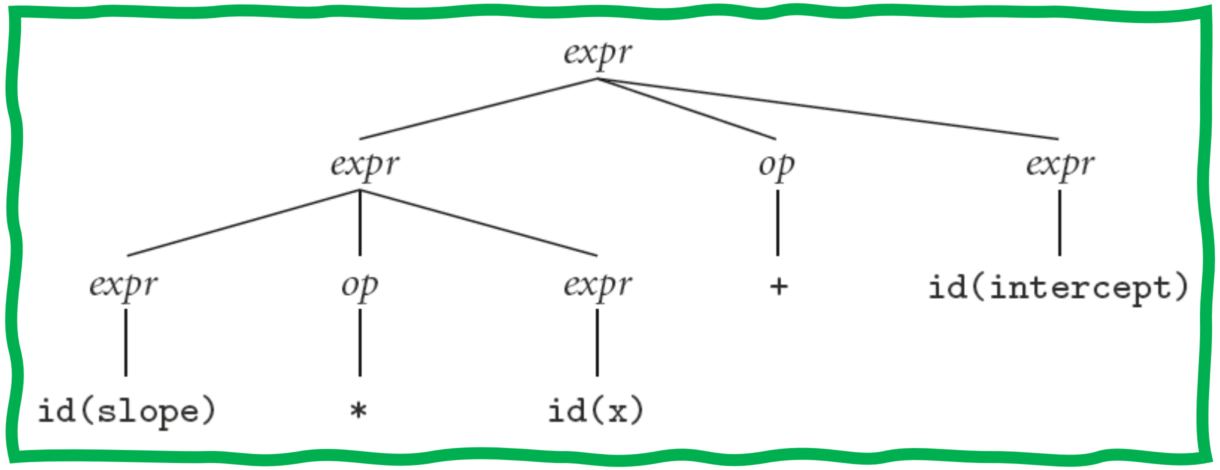


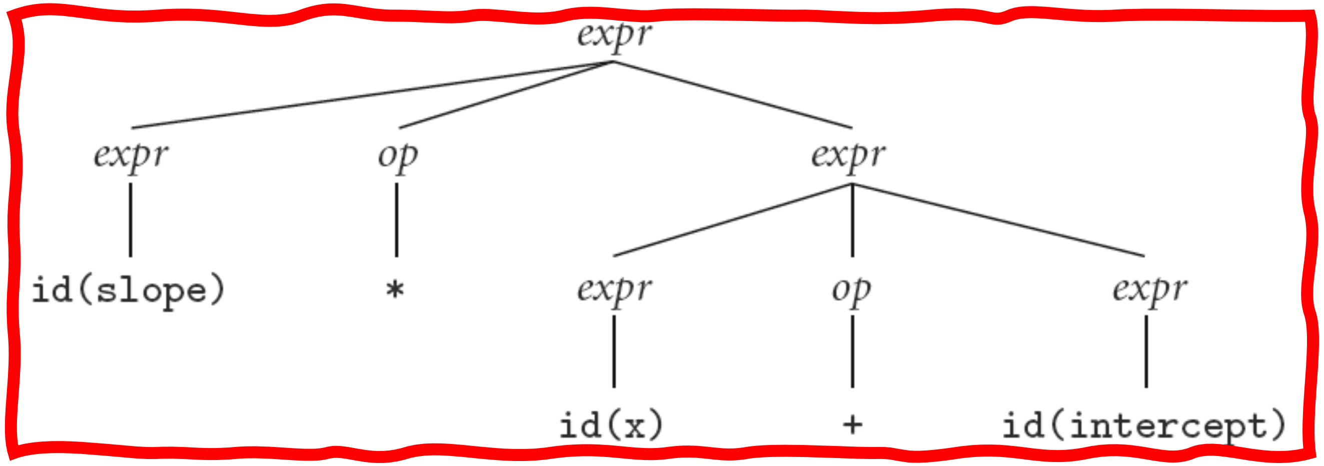
Another parse tree

Context-Free Grammars

$$
\begin{aligned}
expr &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\
&\quad\ \mid expr\ op\ expr \\
op &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

slope * x + intercept





Is there any difference?
Thoughts?

Context-Free Grammars

How about we re-write the grammar as:

1. $expr \longrightarrow term \mid expr\ add\_op\ term$
2. $term \longrightarrow factor \mid term\ mult\_op\ factor$
3. $factor \longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ factor \mid \texttt{(}\ expr\ \texttt{)}$
4. $add\_op \longrightarrow \texttt{+} \mid \texttt{-}$
5. $mult\_op \longrightarrow \texttt{*} \mid \texttt{/}$

Context-Free Grammars

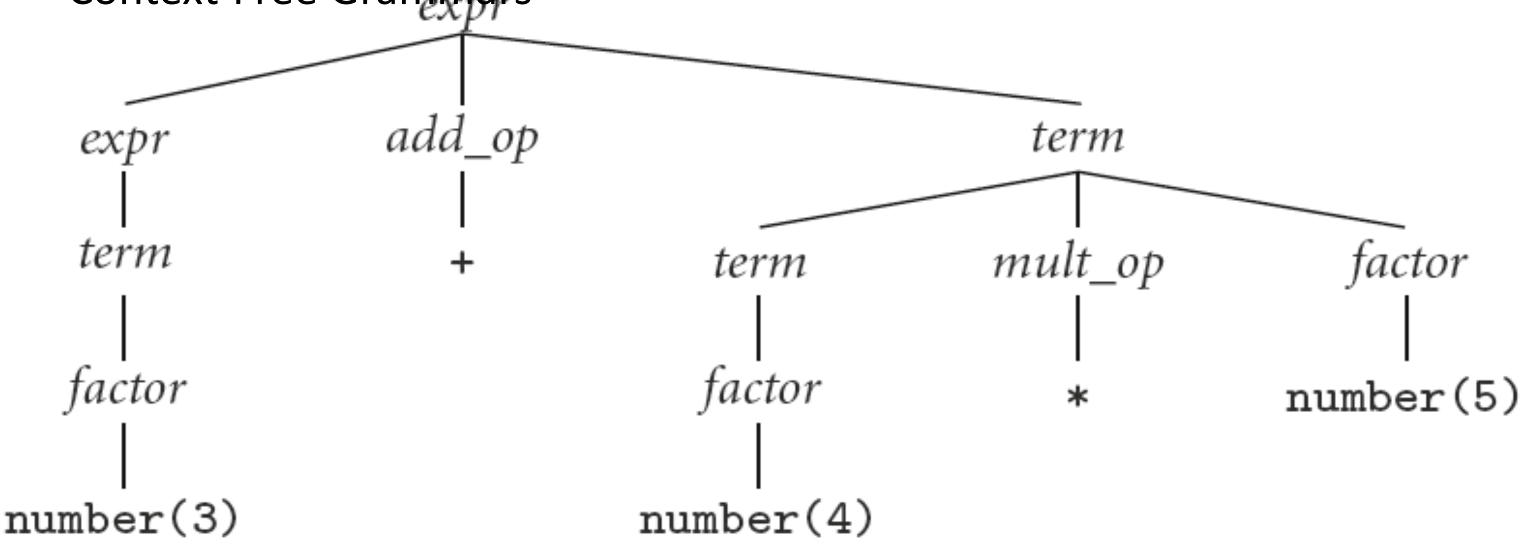
$$
\begin{aligned}
\mathit{expr} &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ \mathit{expr} \mid \texttt{(}\ \mathit{expr}\ \texttt{)} \\
&\qquad \mid \mathit{expr}\ \mathit{op}\ \mathit{expr} \\
\mathit{op} &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

1. $\mathit{expr} \longrightarrow \mathit{term} \mid \mathit{expr}\ \mathit{add\_op}\ \mathit{term}$
2. $\mathit{term} \longrightarrow \mathit{factor} \mid \mathit{term}\ \mathit{mult\_op}\ \mathit{factor}$
3. $\mathit{factor} \longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ \mathit{factor} \mid \texttt{(}\ \mathit{expr}\ \texttt{)}$
4. $\mathit{add\_op} \longrightarrow \texttt{+} \mid \texttt{-}$
5. $\mathit{mult\_op} \longrightarrow \texttt{*} \mid \texttt{/}$

Context-Free Grammars

1. *expr* → *term* | *expr add_op term*
2. *term* → *factor* | *term mult_op factor*
3. *factor* → `id` | `number` | `-` *factor* | `(` *expr* `)`
4. *add_op* → `+` | `-`
5. *mult_op* → `*` | `/`

Context-Free Grammars



Capturing Structure

- Languages can be designed to capture several aspects, for example associativity and precedence
- Associativity: the order in which operators are applied
- Example:

  10 – 4 – 3 is computed as (10 – 4) – 3

  rather than 10 – (4 – 3)
- Precedence: some operators are applied before others
- Example:

  3 + 4 * 5 = 3 + (4 * 5) instead of (3 + 4) * 5

Recaping Terminology

- Context-Free Grammar (CFG)
- Symbols: terminals (tokens) and non-terminals
- Productions
- Derivations (left-most and right-most)
- Parse trees
- Sentential form

Context-Free Grammars

- By analogy to RE and DFAs, a context-free grammar (CFG) is a *generator* for a context-free language (CFL)
  - a parser is a language *recognizer*
- There is an infinite number of grammars for every context-free language
  - not all grammars are created equal, however

Parsing

- We can create parsers that runs in O(n^3) time for any CFG
- Two well-known parsing algorithms that permit this
  - Early's algorithm
  - Cooke-Younger-Kasami (CYK) algorithm
- O(n^3) time is clearly unacceptable for a parser in a compiler - too slow

Parsing

- Fortunately, there are large classes of grammars for which we can build parsers that run in linear time
  - The two most important classes are called **LL** and **LR**
- LL stands for 'Left-to-right, Leftmost derivation'.
- LR stands for 'Left-to-right, Rightmost derivation'

Parsing

- LL parsers are also called 'top-down', or 'predictive' parsers
- LR parsers are also called 'bottom-up', or 'shift-reduce' parsers
- There are several important sub-classes of LR parsers
  - SLR
  - LALR
- We won't be going into detail on the differences between them

Parsing

- Every LL(1) grammar is also LR(1), though right recursion in production tends to require very deep stacks and complicates semantic analysis
- Every CFL that can be parsed deterministically has an SLR(1) grammar (which is LR(1))
- Every deterministic CFL with the *prefix property* (no valid string is a prefix of another valid string) has an LR(0) grammar

Parsing

- Will commonly see LL(K) or LR (K)
  - K indicates how many tokens of look-ahead are required in order to parse
  - Almost all real compilers use one token of look-ahead
- The expression grammar (with precedence and associativity) previously seen is LR(1), but not LL(1)

Parsing

Class	Direction of Scanning	Derivation Discovered	Parse Tree Construction	Algorithms Used
LL	Left-to-right	Left-most	Top-down	Predictive
LR	Left-to-right	Right-most	Bottom-up	Shift-reduce

Main classes of O(n) parsing algorithms

Top-down and Bottom-up Parsing

Go over the string:

“A, B, C;”

With the grammar:

a. id_list → id id_list_tail
b. id_list_tail → , id id_list_tail
c. id_list_tail → ;

LL Parsing

```
An example of an LL(1) grammar:
1.  program          → stmt_list $$
2.  stmt_list        → stmt stmt_list
3.                   | ε
4.  stmt      →      id := expr
5.                   | read id
6.                   | write expr
7.  expr      →      term term_tail
8.  term_tail → add op term term_tail
9.                   | ε
```

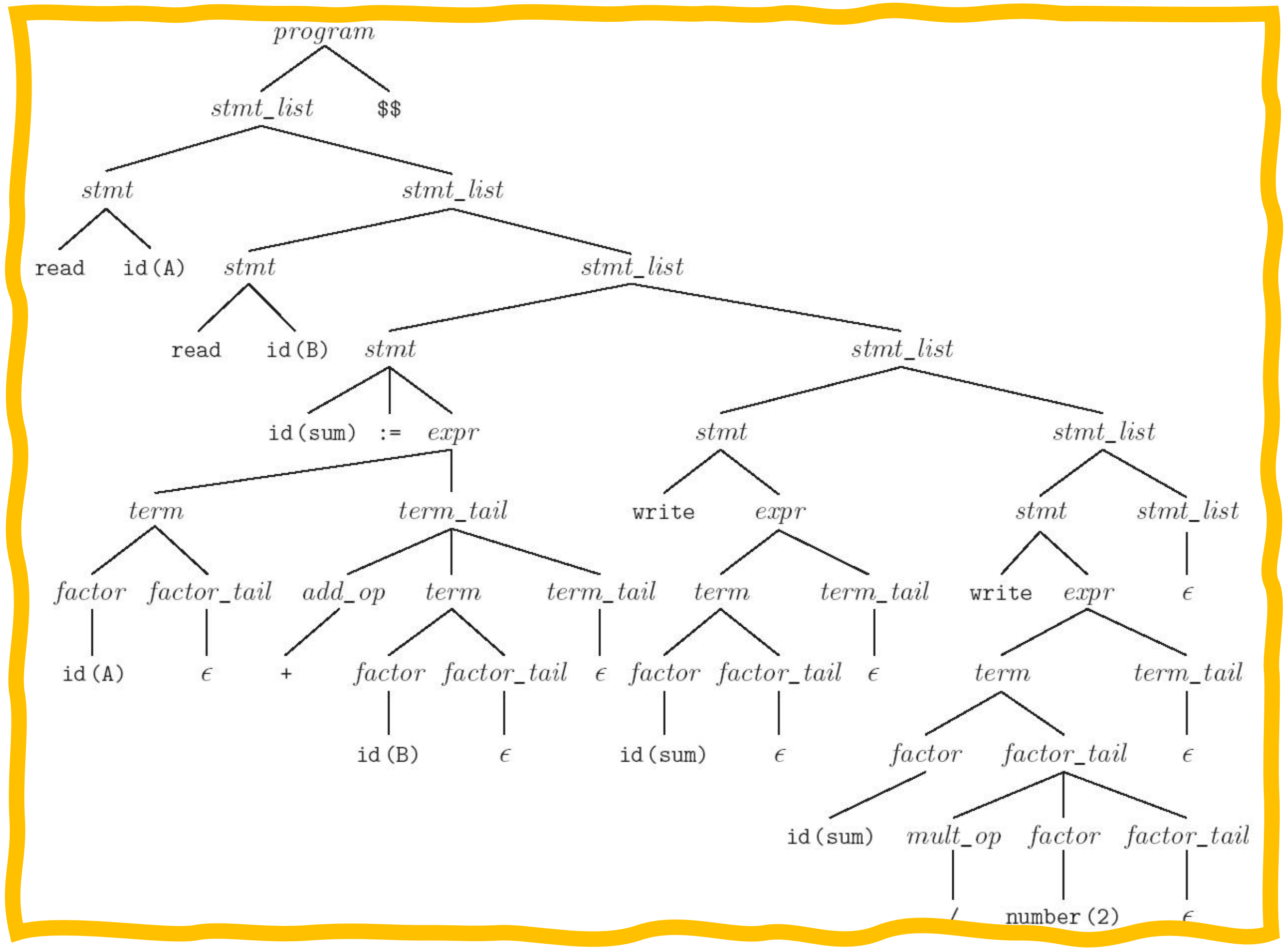
```
10. term      →      factor fact_tail
11. fact_tail → mult_op fact fact_tail
12.                  | ε
13.  factor   →      ( expr )
14.                  | id
15.                  | number
16.  add_op →        +
17.                  | -
18.  mult_op → *
19.                  | /
```

LL Parsing

Consider the program:

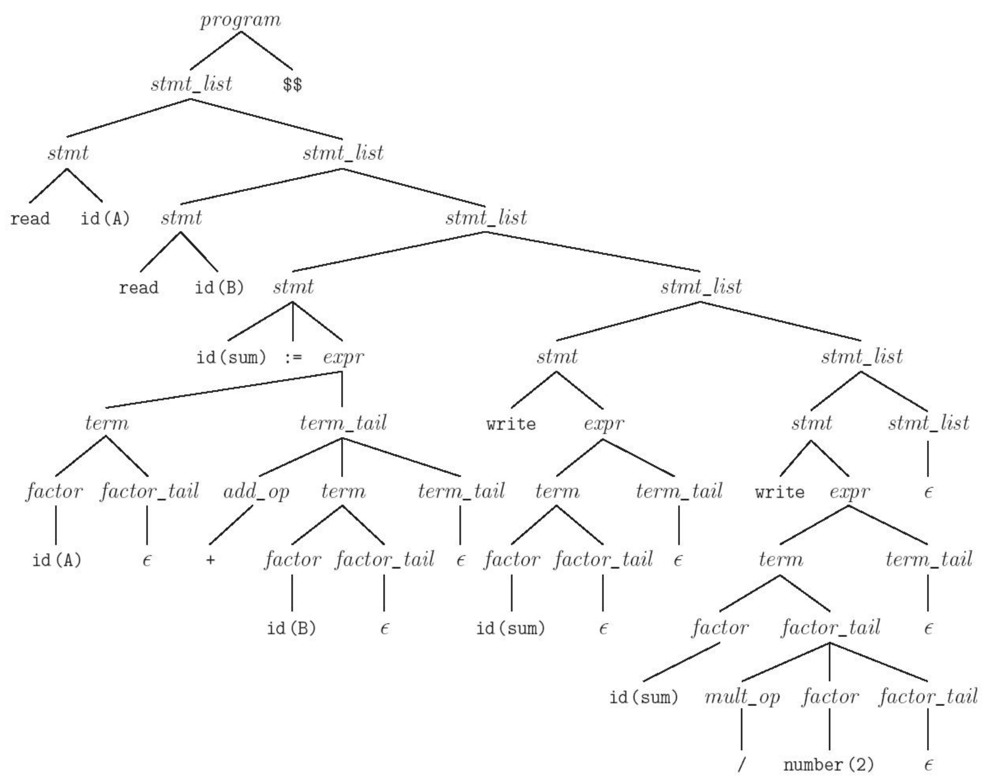
```
read A
read B
sum := A + B
write sum
write sum / 2
```

LL Parsing



program
stmt_list
$$
stmt
stmt_list
read
id(A)
stmt
stmt_list
read
id(B)
stmt
stmt_list
id(sum)
:=
expr
stmt
stmt_list
term
term_tail
write
expr
stmt
stmt_list
factor
factor_tail
add_op
term
term_tail
term
term_tail
write
expr
ϵ
id(A)
ϵ
+
factor
factor_tail
ϵ
factor
factor_tail
ϵ
term
term_tail
id(B)
ϵ
id(sum)
ϵ
factor
factor_tail
ϵ
id(sum)
mult_op
factor
factor_tail
/
number(2)
ϵ

LL Parsing



- Process tokens in a loop
- Repeatedly look up an action in a two-dimensional table based on current leftmost non-terminal and current input token.
- The actions are
  - (1) match a terminal
  - (2) predict a production
  - (3) announce a syntax error

LL Parsing

Top-of-stack	Current input token										
nonterminal	id	number	read	write	:=	(	)	+	-	*	/	$$
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
factor_tail	12	–	12	12	–	–	12	12	12	11	11	12
factor	14	15	–	–	–	13	–	–	–	–	–	–
add_op	–	–	–	–	–	–	–	16	17	–	–	–
mult_op	–	–	–	–	–	–	–	–	–	18	19	–

LL(1) parse table for calculator grammar

LL Parsing

To keep track of the left-most non-terminal, you push the as-yet-unseen portions of productions onto a stack

- for details see Figure 2.21

The key thing to keep in mind is that the stack contains all the stuff you expect to see between now and the end of the program

- what you *predict* you will see

LL Parsing

Some issues with LL parsing:

- Left-recursion
- Common prefixes
- Dangling else

NOTE: eliminating left recursion and common prefixes does NOT make a grammar LL:

  - there are infinitely many non-LL LANGUAGES, and the mechanical transformations work on them just fine
  - the few that arise in practice, however, can generally be handled with kludges

LL Parsing

Fixing left recursion:

```
id_list →    id
        |    id_list , id
```

Becomes

```
id_list      → id id_list_tail
id_list_tail → , id id_list_tail
             | epsilon
```

LL Parsing

- Common prefixes: solved by "left-factoring”
- Example:

```
stmt → id := expr | id ( arg_list )
```

```
equivalently
stmt → id id_stmt_tail
id_stmt_tail → := expr
             | ( arg_list)
```

- we can eliminate left-factor mechanically

LL Parsing

- The"dangling else" problem prevents grammars from being LL(1) (or in fact LL(k) for any k)
- The following natural grammar fragment is ambiguous (Pascal)

```
stmt → if cond then_clause else_clause      |
  other_stuff
then_clause → then stmt
else_clause → else stmt
            | epsilon
```

LL Parsing

The less natural grammar fragment can be parsed bottom-up but not top-down

```
stmt → balanced_stmt | unbalanced_stmt
balanced_stmt → if cond then balanced_stmt
                  else balanced_stmt
              | other_stuff
unbalanced_stmt → if cond then stmt
              | if cond then balanced_stmt
                  else  unbalanced_stmt
```

LL Parsing

Usual fix (top-down OR bottom-up): use the ambiguous grammar together with a *disambiguating rule* that says

- else goes with the closest then or
- more generally, the first of two possible productions is the one to predict (or reduce)

LL Parsing

- Better yet, languages (since Pascal) generally employ explicit end-markers, which eliminate this problem
- In Modula-2, for example, one says:

```
if A = B then
      if C = D then E := F end
else
      G := H
end
```

- Ada says 'end if'; other languages say 'fi'

LL Parsing

- First and Follow algorithm

LL Parsing

```
repeat
  next_sym = stack.pop ()
  if (next_sym in Terminals)
    match (next_sym)      // removes symbol from input and stack
    if (next_sym == $) return ACCEPT
  else if (table[next_sym,next_token].action == ERROR)
    parse_error
  else
    prod = table[next_sym,next_token].prod
    foreach sym in reverse(rhs(prod))
       tack.push (sym)
until stack.top () == $
```

Stack	id	Num	Read	Write	:=	(	)	+	-	*	/	$
prog	1		1	1								1
s_list	2		2	2								3
stmt	4		5	6								-
expr	7	7				7						-
t_tail	9		9	9			9	8	8			9
term	10	10				10						-
f_tail	12		12	12			12	12	12	11	11	12
factor	14	15				13						-
add_op	--							16	17			-
mul_op	--									18	19	-

LL Parsing

Consider the following simpler grammar:

$E \rightarrow T\ E'$

$E' \rightarrow +\ E$

$E' \rightarrow \varepsilon$

$T \rightarrow \text{int}\ T'$

$T \rightarrow (\ E\ )$

$T' \rightarrow *\ T$

$T' \rightarrow \varepsilon$

LL Parsing

Building the FIRST set

- FIRST(a): set of terminals that can start a string of terminals
- T: Set of terminal symbols
- N: Set of non-terminal symbols

Rules:

1. If t in T: First(t) = {t}
2. If X in N and X → ε exists: add ε to First(X)
3. If X in and X → Y_1 Y_2 … Y_m, Y_i in N, then:

```
for i in 1..m:
  if (i = 1 or Y_1…Y_{i-1} is nullable)
     First(x) = First(x) U First(Y_i)
```

LL Parsing

```
foreach t in T
  Eps(t) = false
  First(t) = { t }
foreach X in N
   Eps(X) = if X → ε then true else false
   First(X) = {}
repeat
  foreach productions X → Y1 Y2 … Yk,
    for i in 1..k
      add First(Yi) to First(X)
      if (not Eps(Yi)) continue with foreach loop
      Eps(X) = true
until no further progress
```

Algorithm to build First sets

LL Parsing

foreach symbol X

  Follow(X) = { }

repeat

  foreach production $A \rightarrow \alpha\ B\ \beta$

    $Follow(B) = Follow(B) \cup First(\beta)$

  foreach production $A \rightarrow \alpha\ B$ or $A \rightarrow \alpha\ B\ \beta$ with Eps(b) = true

    $Follow(B) = Follow(B) \cup Follow(A)$

until no further progress

Algorithm to build Follow sets

LL Parsing

- Apply rule (1):

Symbol	First
(	(
)	)
+	+
*	*
int	int
T’	
E’	
T	
E	

LL Parsing

- Apply rule (2), we have E’ → ε and T’ → ε

Symbol	First
(	(
)	)
+	+
*	*
int	int
T’	ε
E’	ε
T	
E	

LL Parsing

Rule (3):

```
3.   If X in and X → Y1 Y2 … Ym, Yi in N, then:
       for i in 1..m:
         if (i = 1 or Y1…Yi-1 is nullable)
           First(x) = First(x) U First(Yi)
```

E → T E'
E' → + E
E' → ε
T → int T'
T → (E)
T' → * T
T' → ε

E → T E
First(E) = First(E) U First(T)

Computing First(T) = {int, (} (By rule a)

So First(E) = First(T)

We don't include First(E') because T is not nullable

Symbol	First
(	(
)	)
+	+
*	*
int	int
T'	ε
E'	ε
T	{int, (}
E	{int, (}

LL Parsing

Rule (3):

```
3.   If X in and X → Y1 Y2 … Ym, Yi in N, then:
       for i in 1..m:
         if (i = 1 or Y1…Yi-1 is nullable)
            First(x) = First(x) U First(Yi)
```

E → T E'
E' → + E
E' → ε
T → int T'
T → (E)
T' → * T
T' → ε

First (E') = { ε, + }

Symbol	First
(	(
)	)
+	+
*	*
int	int
T'	ε
E'	ε, +
T	int, (
E	int, (

LL Parsing

Rule (3):

3. If X in and X → Y_1 Y_2 … Y_m, Y_i in N, then:
   for i in 1..m:
     if (i = 1 or $Y_1 \ldots Y_{i-1}$ is nullable)
       First(x) = First(x) U First(Y_i)

E → T E'
E' → + E
E' → ε
T → int T'
T → (E)
T' → * T
T' → e

First (T') = { ε, * }

Symbol	First
(	(
)	)
+	+
*	*
int	int
T'	ε, *
E'	ε, +
T	int, (
E	int, (

LL Parsing

Computing the Follow set of non-terminals

1. If \$ is the input end-marker, and S is the start symbol, $\$ \in \text{Follow}(S)$.
2. If there is a production, $A \rightarrow \alpha B \beta$, then $(\text{FIRST}(\beta) - \varepsilon) \subseteq \text{Follow}(B)$.
3. If there is a production, $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where $\varepsilon \in \text{FIRST}(\beta)$, then $\text{Follow}(A) \subseteq \text{Follow}(B)$.

LL Parsing

Computing the Follow set of non-terminals

1. If $ is the input end-marker, and S is the start symbol, $ ∈ FOLLOW(S).

→ Include $ in Follow(S)

E → T E'
E' → + E
E' → ε
T → int T'
T → (E)
T' → * T
T' → e

LL Parsing

2. If there is a production, $A \rightarrow \alpha B\beta$, then $(\text{FIRST}(\beta) - \varepsilon) \subseteq \text{FOLLOW}(B)$.

We look at the occurrence of a non-terminal on the right-hand side of a production which is followed by something

2a. E → T . E'

Follow(T) contains First(E') = {ε,+} – {ε} = {+}

2b. T → (E .)

Follow(E) contains (at least) First()) = {) }

Follow(E) now becomes {$,) }

LL Parsing

a. E→T X .
   Follow(X) contains (at least) Follow(E) = {), $ }   (from step 2b)
b. ε in First(X) so:
   Follow(T) contains (at least) Follow(E) = {), $, +} (from step 2a and 2b)
c. X→+ E .
   Follow(E) contains (at least) Follow(X) = {), $ } (from step 3a)
d. T → int Y .
   Follow(Y) contains (at least) Follow(T) = {), $, +}. (from step 3b)
e. Y→*T .
   Follow(T) contains (at least) Follow(Y) = {), $ } (from step 3d)

LL Parsing

We do this whole process again until no more additions happen:

f. E→T X .
   Follow(X) contains (at least) Follow(E) = {), $ }   (no change)

g. ε in First(X) so:
   Follow(T) contains (at least) Follow(E) = {), $, +} (no change)

h. X→+ E .
   Follow(E) contains (at least) Follow(X) = {), $ } (no change)

i. T → int Y .
   Follow(Y) contains (at least) Follow(T) = {), $, +} (no change)

j. Y→*T .
   Follow(T) contains (at least) Follow(Y) = {), $ } (no change)

LL Parsing

Symbol	First	Follow
(	(	N/A
)	)	
+	+	
*	*	
int	int	
T’	ε, *	), $, +
E’	ε, +	), $
T	int, (	), $, +
E	int, (	), $

Final First and Follow sets

LR Parsing

- Builds a forest of partial subtrees of the parse tree
- Performs a replacement of the form $Y_1\ Y_2 \ldots Y_k \rightarrow A$ when the production $A \rightarrow Y_1\ Y_2 \ldots Y_k$ is recognized
- Table driven
- Roots of partially recognized sub-trees in stack
- Produce reversed right-most (canonical) derivations

LR Parsing

- Shift action: performed when new token is found in the input, it’s shifted to the stack
- Reduce action: performed when the right—hand side of a production is recognized, pop symbols from stack and insert non-terminal of the left-hand side of the production
- Role of stack is the main difference between LL and LR parsing:
    - Top-down parsing: stack contains symbols that expects to see in the future
    - Bottom-up parsing: stack contains symbols of what the parsing already has seen

LR Parsing

Grammar for list of identifiers:

- id_list → ID id_list_tail
- id_list_tail → , ID id_list_tail
- id_list_tail → ;

Stack Contents (Sentential Forms)	Remaining Input String
ε	A, B, C;
ID (A)	, B, C;
ID (A),	B, C;
ID (A), ID (B)	, C;
ID (A), ID (B),	C;
ID (A), ID (B), ID (C)	;
ID (A), ID (B), ID (C) ;	
ID (A), ID (B), ID (C) id_list_tail	
ID (A), ID (B) id_list_tail	
ID (A) id_list_tail	
id_list	

Symbols in right-hand side of production that are being replaced by the left-hand side (non-terminal) constitute the handle of the sentential form

LR Parsing

1. e → t e’
2. e’ → + t e’
3. e’ → ε
4. t → f t’
5. t’ → * f
6. t’ → ε
7. f → (e)
8. f → ID
9. f → NUM

1. e → e + t
2. e → t
3. t → t * f
4. t → f
5. f → (e)
6. f → ID
7. f → NUM

- Left recursion allows the parser to collapse lists as it goes along
- Left recursion also captures left associativity while keeping operators and operands together in the same right-hand side
- This wasn’t possible in top-down grammars

LR Parsing

- Key: keeping track of the set of productions on which the parser might be, and the position within the productions
- At the beginning, the stack's parser is empty; our state is at the beginning of the start non-terminal symbol
- We represent the position in some production (i.e. the location associated to the top of the parser stack) with a dot symbol (.) in some position of the right-hand side of a production:

$$A \rightarrow .\ B\ \beta$$

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

LR Parsing

- When augmented with a (.) a production becomes an LR item
- The LR item tells us in which production we can be, and it what position:

1. e → . e + t	1. e → e + t	1. e → e + t	1. e → e + t	1. e → e + t
2. e → t	2. e → . t	2. e → t	2. e → t	2. e → t
3. t → t * f	3. t → t * f	3. t → . t * f	3. t → t * f	3. t → t * f
4. t → f	4. t → f	4. t → f	4. t → . f	4. t → f
5. f → (e)	5. f → (e)	5. f → (e)	5. f → (e)	5. f → . (e)
6. f → ID	6. f → ID	6. f → ID	6. f → ID	6. f → . ID
7. f → NUM	7. f → NUM	7. f → NUM	7. f → NUM	7. f → . NUM

LR Parsing

1. e → . e + t
2. e → t
3. t → t * f
4. t → f
5. f → (e)
6. f → ID
7. f → NUM

The original item is the *basis* of the list

Remaining items are the *closure*:

1. e → e + t
2. e → . t
3. t → t * f
4. t → f
5. f → (e)
6. f → ID
7. f → NUM

1. e → e + t
2. e → t
3. t → . t * f
4. t → f
5. f → (e)
6. f → ID
7. f → NUM

1. e → e + t
2. e → t
3. t → t * f
4. t → . f
5. f → (e)
6. f → ID
7. f → NUM

1. e → e + t
2. e → t
3. t → t * f
4. t → f
5. f → . (e)
6. f → . ID
7. f → . NUM

LR Parsing

1. e → . e + t
2. e → . t
3. t → . t * f
4. t → . f
5. f → . (e)
6. f → . ID
7. f → . NUM

- The list of LR items represents the state of the parser
- By shifting and reducing, the set of items will change
- If a state in which some item has the (.) at the end of the right-hand side of a production, we reduce by that production
- If we need to shift, but the next token cannot follow the (.) in any item of the current state, then we detect a syntax error

LR Parsing

- Suppose the input string is: ID * NUM + ID
- Next token is ID, which is shifted into the stack
- Our options are now narrowed down to #6 ➔ single basis and empty closure
- Since the (.) is at the end #6, we must reduce

1. e → . e + t
2. e → . t
3. t → . t * f
4. t → . f
5. f → . (e)
6. f → . ID
7. f → . NUM

Stack: $

6. f → ID .

Stack: $ ID

3. t → t * f .
4. t → f .

Stack: $ f

LR Parsing

- Remaining input string is: * NUM + ID
- We could be either in #3 or #4
- (.) is at the end of #4, so reduce again
- Next token is *, so we shift it into the stack

3. t → t * f
4. t → f .

Stack: $ f

3. t → t . * f
4. t → f

Stack: $ t

3. t → t * . f

Stack: $ t *

LR Parsing

- Remaining input string is: NUM + ID
- We could be either in #3, #5, #6 or 7
- Next token is NUM, we should then be in #7
- Shift token from input to stack

3. t → t * . f
5. f → . (e)
6. f → . ID
7. f → . NUM

Stack: $ t *

3. t → t * f
5. f → (e)
6. f → ID
7. f → NUM .

Stack: $ t * ID

3. t → t * f .

Stack: $ t * f

1. e → e + t .
2. e → t .

Stack: $ t

LR Parsing

- Remaining input string is: + ID
- We have reached the end of #2: reduce
- Next token is +, shift it to stack
- New item list mark the beginning with t

First item is the basis; remaining items form the closure

1. e → e + t
2. e → t .

Stack: $ t

1. e → e . + t
2. e → t

Stack: $ e

1. e → e + . t
2. e → t

Stack: $ e +

1. e → e + . t
3. t → . t * f
4. t → . f
5. f → . (e)
6. f → . ID
7. f → . NUM

Stack: $ e +

LR Parsing

- Remaining input string is ID, we shift it
- Reduce ID by f
- Reduce f by t
- Reduce e + t by e

6. f → . ID	6. f → ID .	4. t → f .	1. e → e + t .	4. e → t .
Stack: $ e +	Stack: $ e + ID	Stack: $ e + f	Stack: $ e + t	Stack: $ e

LR Parsing

- Shift rules work as transition functions in the automaton
- Each state of the automaton corresponds to a list of items
- Set of items represent where we can be at some point in the parsing process
- Sets of items constructed during the parser construction, but not needed during actual parsing

LR Parsing

Family of parsers:

- LR(0): cannot handle shift-reduce conflicts
- SLR(1) or Simple LR: parser peeks at input and use Follow sets to resolve conflicts (reduction $A \rightarrow \alpha$ is applied if the next token is in Follow(α))

Issues arise when the token is also in the First set of the symbols following a (.)

- LALR(1):
  - Use state-specific look-ahead to disambiguate
  - Most common parsers; resolve more conflicts
- Full LR:
  - Much larger and complex than LALR and
  - Duplicates states

LR Parsing

LR Parsing algorithm:

```
loop
  state = stack.top ()
  if (state == start_state and sym == start_sym)
     accept
```