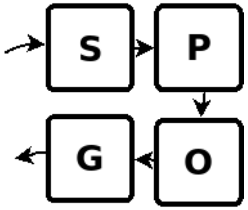
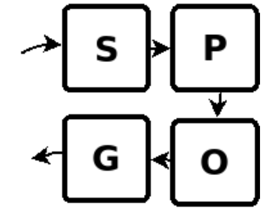


Parsing and Context Free Grammars (Chapter 4)

Copyright © 2019 Jonathan Cook



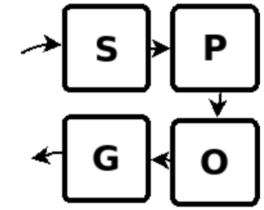
Recall: now we have a scanner that does lexical analysis for us, giving us a token sequence rather than a character sequence.

Now, we need to match the token sequence to our programming language's grammar.

Or, we need to **parse** it.

As we parse it, we create data that represents our program, and then use that in the back end to generate our output language (e.g., object code or JVM bytecode or assembly code or...)

Copyright © 2019 Jonathan Cook



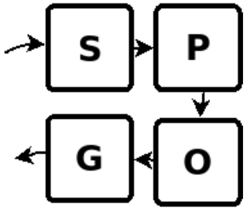
Recall: for top-down recursive descent parsing, we used a **context-free grammar** (CFG) to specify our input language.

Some form of CFGs are **the canonical way** to specify programming languages…

Even though the CFG cannot capture everything that specifies a compile-able program!

E.g.: A CFG cannot encode that a variable must be declared before it is used.

Copyright © 2019 Jonathan Cook



Formalities of Context Free Grammars (textbook sec 4.2.1)

A CFG is a 4-tuple < T, N, s, R > where:

T is the set of terminals (token types)
N is the set of non-terminals
$s \in N$ is the starting non-terminal
R is the set of production rules
  $r \in R$, $r = n \rightarrow v$, $n \in N$, $v \in (T \cup N)^*$

Copyright © 2019 Jonathan Cook

CFG:

S → A B A
A → a
B → ε
B → b B

This grammar accepts strings like aba, abba, …

IOW, its language is the same as the regular expression ab*a

Indeed, CFGs can encode any language that REs can…and more!

Copyright © 2019 Jonathan Cook

REs define *regular languages*

CFGs define *context-free languages*

So, all regular languages *are* context-free languages!

Does this mean there is a language hierarchy?

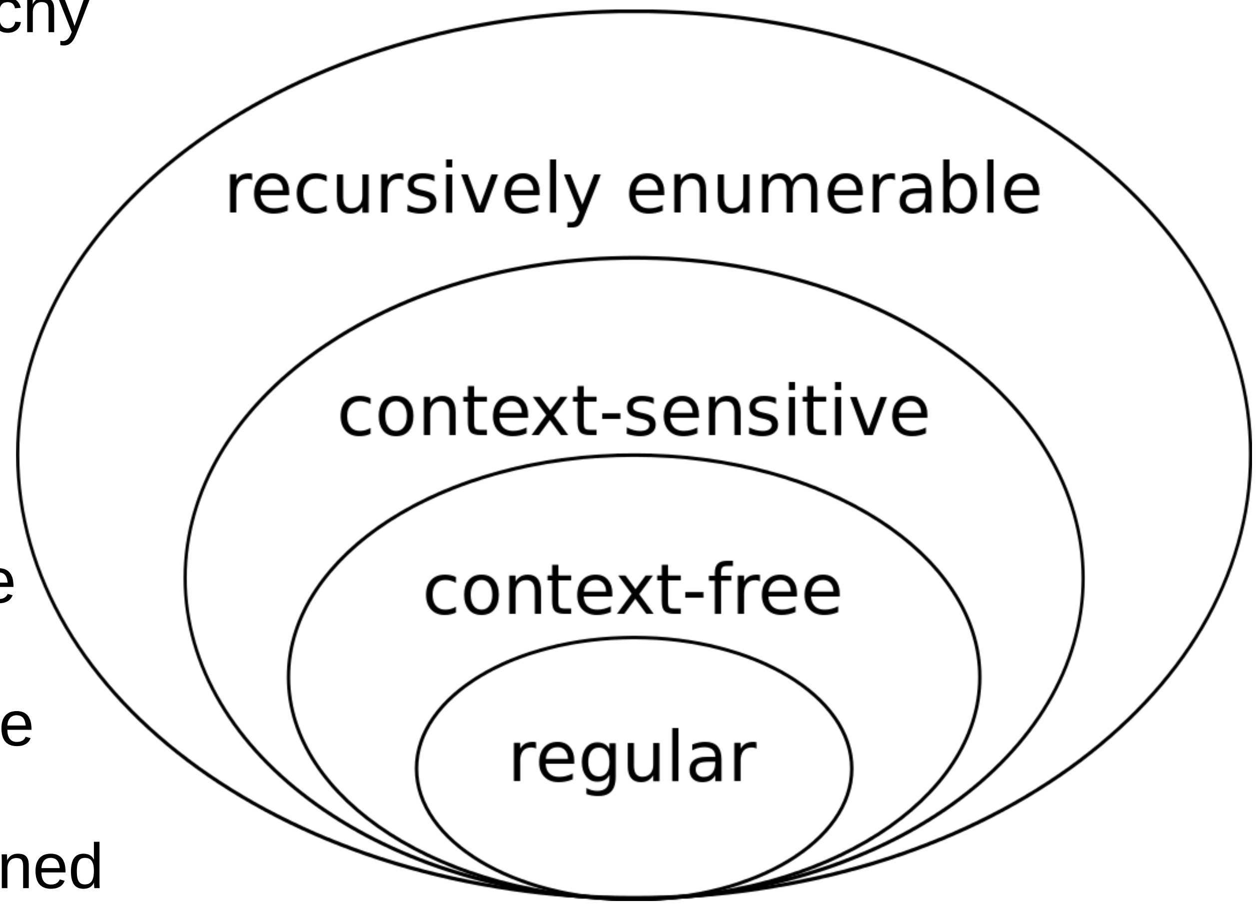
YES!

Copyright © 2019 Jonathan Cook

Chomsky Hierarchy of Languages

Wikipedia:Chomsky_hierarchy

Notes
- CF: the nonterminal head of the production rule has no context
- CS: head allows more than just the nonterminal
- RecEnum: think “can write a no-input function to enumerate all strings in the language”
- These four have been refined over the years



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

Copyright © 2019 Jonathan Cook

Textbook Symbol Usage

Uppercase letters are non-terminals: A,B,...

Lowercase letters are terminals: a,b,…

Lowercase Greek letters are sequences of terminals and nonterminals: α, β, ….

Copyright © 2019 Jonathan Cook

Example Expression Grammar in textbook p198

Expr → Expr ‘+’ Term
Expr → Expr ‘-’ Term
Expr → Term
Term → Term * Factor
Term → Term / Factor
Term → Factor
Factor → ‘(‘ Expr ‘)’
Factor → id
Factor → number

or

E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | id | number

Copyright © 2019 Jonathan Cook

$E \rightarrow E+T \mid E-T \mid T$
$T \rightarrow T*F \mid T/F \mid F$
$F \rightarrow (E) \mid id \mid number$

string: “i*42-j” (i,j are id’s, 42 is a number)

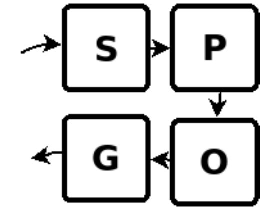
derivation:
$E \Rightarrow E-T \Rightarrow T-T \Rightarrow T*F-T \Rightarrow F*F-T \Rightarrow id*F-T \Rightarrow$
$id*number-T \Rightarrow id*number-F \Rightarrow id*number-id$

Each derivation step gives a **sentential form** of G

If $S \Rightarrow^* \alpha$, where S is the starting nonterminal and α is a sequence of nonterminals and terminals, then α is a sentential form

A sentential form without any nonterminals is a **sentence** of G, or a sentence in L(G)

Copyright © 2019 Jonathan Cook



E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | id | number

string: “i*42-j” (i,j are id’s, 42 is a number)

E ⇒ E-T ⇒ T-T ⇒ T*F-T ⇒ F*F-T ⇒ id*F-T ⇒ id*number-T ⇒ id*number-F ⇒ id*number-id

leftmost derivation: the leftmost nonterminal in each sentential form is always chosen for the next derivation; the above is a leftmost derivation; each step is a **left sentential form**

rightmost derivation: rightmost nonterminal is chosen; see below

E ⇒ E-T ⇒ E-F ⇒ E-id ⇒ T-id ⇒ T*F-id ⇒ T*number-id ⇒ F*number-id ⇒ id*number-id

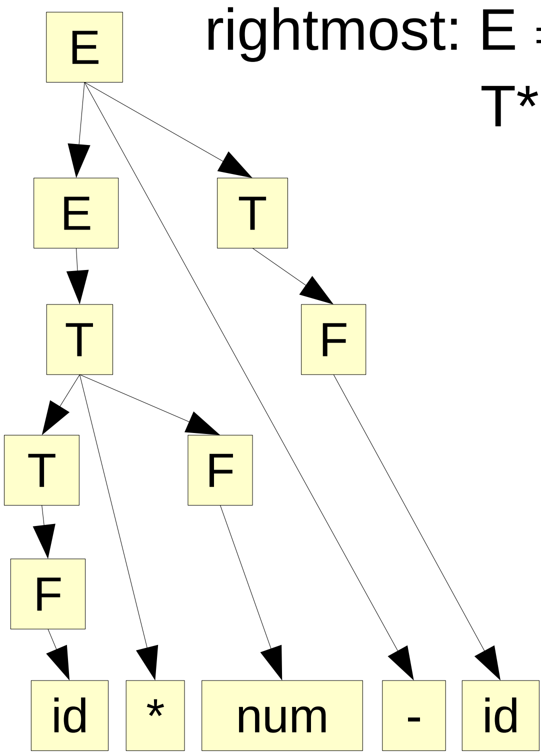
Copyright © 2019 Jonathan Cook

$E \rightarrow E{+}T \mid E{-}T \mid T$
$T \rightarrow T{*}F \mid T/F \mid F$
$F \rightarrow (E) \mid id \mid number$

string: “i*42-j” (i,j are id’s, 42 is a number)

leftmost: $E \Rightarrow E\text{-}T \Rightarrow T\text{-}T \Rightarrow T{*}F\text{-}T \Rightarrow F{*}F\text{-}T \Rightarrow id{*}F\text{-}T \Rightarrow$
$id{*}number\text{-}T \Rightarrow id{*}number\text{-}F \Rightarrow id{*}number\text{-}id$

rightmost: $E \Rightarrow E\text{-}T \Rightarrow E\text{-}F \Rightarrow E\text{-}id \Rightarrow T\text{-}id \Rightarrow T{*}F\text{-}id \Rightarrow$
$T{*}number\text{-}id \Rightarrow F{*}number\text{-}id \Rightarrow id{*}number\text{-}id$



Notes:
- both derivations produce the same parse tree!
- this is because this grammar is unambiguous!

Copyright © 2019 Jonathan Cook

Eliminating *immediate* left recursion

with production rules like

A → Aa | Ab | Ac | q | r | s

then rewrite with new nonterminal B like

A → qB | rB | sB
B → aB | bB | cB | empty

(a-c) and (q-s) are *examples*; these could be *sequences* of terminals and nonterminals

bottom of page 212 of textbook

Copyright © 2019 Jonathan Cook

Eliminating *ALL* left recursion

rules might have left recursion that refers to other nonterminals, not just itself; this is non-immediate left recursion. How to remove? Algorithm 4.19 p 213 of tbook

Informally, given recursion of the form

$A \rightarrow Bq$ (r, s, q are sequences of terms and non-terms)
$B \rightarrow r \mid s$ (r and s have A somewhere in them)

Replace rules for A by expanding B rules into them, as:

$A \rightarrow rq \mid sq$

And then eliminate any immediate left recursion that might exist for A.

Repeat for all production rules (real alg imposes ordering and applies above only to previously-seen nonterminals)

Copyright © 2019 Jonathan Cook

Eliminating *ALL* left recursion

Example grammar:

$S \rightarrow A\ a \mid b$

$A \rightarrow A\ c \mid S\ d \mid \varepsilon$

order: S,A; nothing to do for S (no previous nt’s); at A, we replace S with its productions, so that we get

$S \rightarrow A\ a \mid b$

$A \rightarrow A\ c \mid A\ a\ d \mid b\ d \mid \varepsilon$

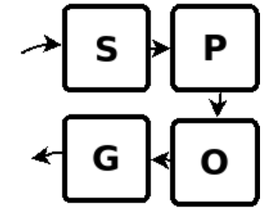
Now we eliminate the immediate left recursion to get

$S \rightarrow A\ a \mid b$

$A \rightarrow b\ d\ B \mid B$

$B \rightarrow c\ B \mid a\ d\ B \mid \varepsilon$

Copyright © 2019 Jonathan Cook



Left Factoring

Left factoring is a simple transformation that just removes equal production rule prefixes from the rules so that top down parsing can handle them

Example grammar:
S → A b | A c
A → a A | b A | ε

Rules for S both begin with A, so cannot immediately choose.
Left factoring just removes the suffixes into another nonterm:

S → A B
B = b | c
A → a A | b A | ε

Now only one rule for S that begins with prefix A. In general, any prefix (string of terminals and nonterminals) can be factored.

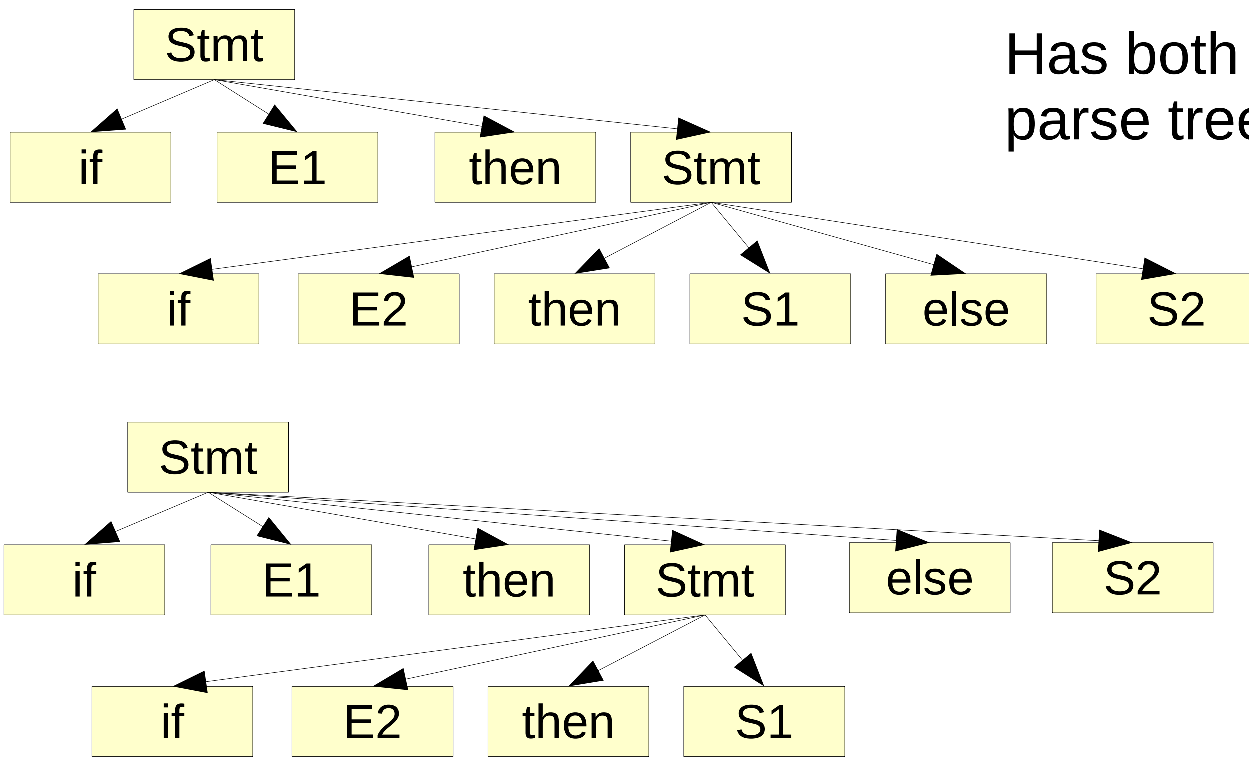
Copyright © 2019 Jonathan Cook

Ambiguity: if-then-else example

Example grammar:

Stmt → ‘if’ Expr ‘then’ Stmt
　　| ‘if’ Expr ‘then’ Stmt ‘else’ Stmt
　　| // other statements

Input token sequence: if E1 then if E2 then S1 else S2



Has both of these possible parse trees! We want the first!

Real C code:

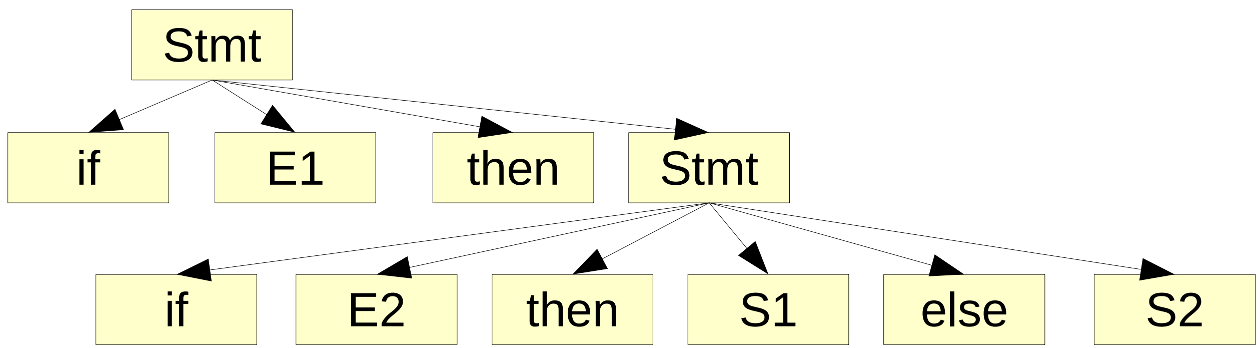
```
if (x>2)
   if (x>5)
      x = 11;
   else
      x = 42;
```

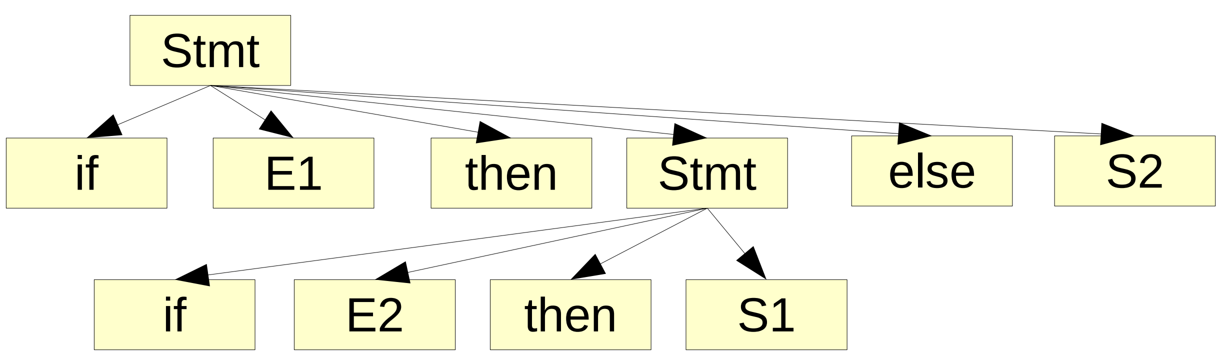
Copyright © 2019 Jonathan Cook

Ambiguity: if-then-else example

Example grammar:
Stmt → ‘if’ Expr ‘then’ Stmt
 | ‘if’ Expr ‘then’ Stmt ‘else’ Stmt
 | <other statements>

“if E1 then if E2 then S1 else S2”





Textbook, p211: “In all programming languages with conditional statements of this form, the first parse tree is preferred. The general rule is, “Match each *else* with the closest unmatched *if*. This disambiguation rule can theoretically be incorporated directly into the grammar, but in practice it is rarely built into productions.”

Copyright © 2019 Jonathan Cook

Top Down Parsing (TB Sec 4.4)

We looked at simple top-down parsing in Chapter 2, using recursive descent parsing
- and saw that left recursion can be a problem

But we didn't think too hard about what works and what doesn't for top-down parsing

Example grammar:
$S \rightarrow A\ S \mid B\ S$
$A \rightarrow a\ B$
$B \rightarrow b$

In top-down, we start at S, but with no terminal symbol in any production rule, we don't have an immediate choice

But, if we look ahead at the waiting token, it will clearly tell us whether to recurse into nontermA() or nontermB().

Copyright © 2019 Jonathan Cook

Top Down Parsing (TB Sec 4.4)

We can determine if a grammar is suitable for top-down.
Helper definitions:

FIRST(α) = the set of terminals that begin all strings derivable from α, where α is a sequence of grammar symbols (terminals and nonterminals).

FOLLOW(A) = the set of terminals that can immediately follow the nonterminal A in any sentential form

Example grammar:

$S \rightarrow A\ S \mid B\ S \mid \varepsilon$

$A \rightarrow a\ B$

$B \rightarrow b$

FIRST(A S) = {a}, FIRST(B S) = {b}, FIRST(ε) = {ε}
FOLLOW(S) = {$}
FIRST(a B) = {a}, FOLLOW(A) = {a,b,$}
FIRST(b) = {b}, FOLLOW(B) = {a,b,$}

Copyright © 2019 Jonathan Cook

LL(1) Grammars (TB Sec 4.4)

If a grammar is parse-able top-down (no backtracking) by only looking at the next input token, then it is an **LL(1) grammar**:
L == scanning input Left-to-right
L == producing a Leftmost derivation
1 == looking ahead only 1 input token

All LL(1) grammars are context-free, but not all CFGs are LL(1)

Example grammar:

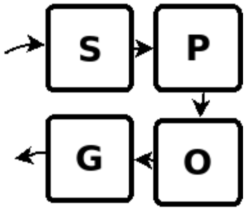
$S \rightarrow A\ S \mid B\ S \mid \varepsilon$

$A \rightarrow a\ B$

$B \rightarrow b$

FIRST(A S) = {a}, FIRST(B S) = {b}, FIRST(ε) = {ε}
FOLLOW(S) = {$}
FIRST(a B) = {a}, FOLLOW(A) = {a,b,$}
FIRST(b) = {b}, FOLLOW(B) = {a,b,$}

Copyright © 2019 Jonathan Cook



LL(1) Grammars (TB Sec 4.4)

Requirements (p224):
- FIRST(α) is disjoint for all production rules of a nonterminal S
- if one production rule of a nonterminal S produces ε, then FIRST()'s of all S rules are disjoint from FOLLOW(S)

All LL(1) grammars are context-free; not all CFGs are LL(1)

Example grammar: Yes, this grammar is LL(1)

$S \rightarrow A\ S \mid B\ S \mid \varepsilon$

$A \rightarrow a\ B$

$B \rightarrow b$

FIRST(A S) = {a}, FIRST(B S) = {b}, FIRST(ε) = {ε}

FOLLOW(S) = {$}

FIRST(a B) = {a}, FOLLOW(A) = {a,b,$}

FIRST(b) = {b}, FOLLOW(B) = {a,b,$}

Copyright © 2019 Jonathan Cook

LL(1) Grammars (TB Sec 4.4)

Top-down parsing does not **have** to be done recursively!

Algorithm 4.31 (p224) builds a parsing table for an LL(1) grammar, and parsing can be table-driven (Alg 4.34)

S → A S | B S | ε
A → a B
B → b
FIRST(A S) = {a}, FIRST(B S) = {b}, FIRST(ε) = {ε}
FOLLOW(S) = {$}
FIRST(a B) = {a}, FOLLOW(A) = {a,b,$}
FIRST(b) = {b}, FOLLOW(B) = {a,b,$}

NT\LA	a	b	$
S	S→A S	S→B S	S->ε
A	A →a B		
B		B→b	

Copyright © 2019 Jonathan Cook

LL(1) Grammars (TB Sec 4.4)

S → A S | B S | ε
A → a B
B → b

NT\LA	a	b	$
S	S→A S	S→B S	S->ε
A	A →a B		
B		B→b	

Algorithm --
Repeat until match $:
- if top is terminal, match and consume, or err
- if top is nonterminal, use lookahead to select table rule, replace on stack with production rule RHS

Parsing string "ab":
1. stack = [S $], lookahead = 'a' (replace S with A S on stack)
2. stack = [A S $], lookahead = 'a' (replace A with a B)
3. stack = [a B S $], lookahead = 'a' (now match & consume)
4. stack = [B S $], lookahead = 'b' (replace B with b)
5. stack = [b S $], lookahead = 'b' (now match & consume)
6. stack = [S $], lookahead = '$' (replace S with (empty))
7. stack = [$], lookahead = '$' (match and finish)

Copyright © 2019 Jonathan Cook

Bottom Up Parsing (TB Sec 4.5)

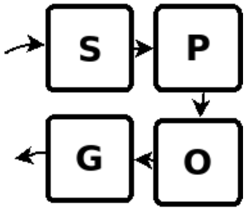
LL(1) grammars can be parsed with top down parsing with one-token lookahead. Great! Note that **this is really used** in the **ANTLR** parser generator tool (which is Java based). ANTLR actually uses "LL(*)" parsing.

However, **yacc** does bottom up parsing. So we are going to spend time learning bottom up parsing, in more detail than we did top down parsing!

TB p234: "We can think of bottom up parsing as the process of 'reducing' a string w to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production."

Some of you may have seen a "shift/reduce" conflict warning in your yacc grammars; the "reduce" word in this warning is the "reduction step" in the textbook quote above.

Copyright © 2019 Jonathan Cook



Bottom Up Parsing (TB Sec 4.5)

E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | id | number

string: “i*42-j” (i,j are id’s, 42 is a number)

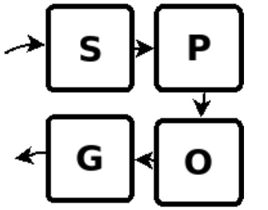
bottom-up: id*number-id ⇒ F*number-id ⇒ T*number-id ⇒
T*F-id ⇒ T-id ⇒ E-id ⇒ E-F ⇒ E-T ⇒ E

If you reverse the sequence above, you get the rightmost derivation! (leftmost and rightmost shown below from earlier)

leftmost: E ⇒ E-T ⇒ T-T ⇒ T*F-T ⇒ F*F-T ⇒ id*F-T ⇒
id*number-T ⇒ id*number-F ⇒ id*number-id

rightmost: E ⇒ E-T ⇒ E-F ⇒ E-id ⇒ T-id ⇒ T*F-id ⇒
T*number-id ⇒ F*number-id ⇒ id*number-id

Copyright © 2019 Jonathan Cook



Bottom Up Parsing (TB Sec 4.5)

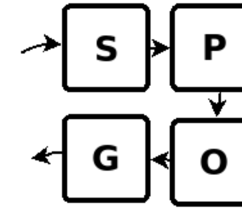
E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | id | number

string: “i*42-j” (i,j are id’s, 42 is a number)

bottom-up: id*number-id ⇒ F*number-id ⇒ T*number-id ⇒ T*F-id ⇒ T-id ⇒ E-id ⇒ E-F ⇒ E-T ⇒ E

Note:
- scanning of input still proceeds Left to right
- but we produced a Rightmost derivation
- so we can call this **LR** parsing, and just like LL(k) grammars there are LR(k) grammars

Copyright © 2019 Jonathan Cook



Bottom Up Parsing (TB Sec 4.5)

How to do bottom up parsing? First, definitions

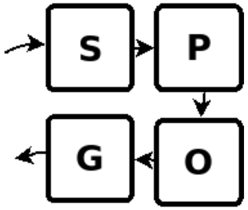
Handle: “a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.” (TB p235)

If $S \rightarrow^* \alpha A \omega \rightarrow \alpha\beta\omega$ in a rightmost derivation, then β is a handle in position following α (TB p235) (ω must have only terminals)

Note: Handles are contextual, not generic (see slide bottom)
Note: Not all leftmost strings that match a production rule must be handles – only those that would be in a rightmost derivation!

“If a grammar is unambiguous, then every right sentential form of the grammar has exactly one handle” (TB, p235)

Copyright © 2019 Jonathan Cook



Bottom Up Parsing (TB Sec 4.5)

How to do bottom up parsing? **Shift-Reduce** parsing

Again (as in LL parsing), use a stack.

Shift (push) input symbols (tokens) onto stack until the top of the stack contains a **handle**.

Reduce the handle to its production rule head.

Go back to shifting input.

If at end of input stack contains only the starting nonterminal, accept; else report error

Copyright © 2019 Jonathan Cook

Bottom Up Parsing (TB Sec 4.5)

E → E+T | E-T | T   string: “i*42-j”
T → T*F | T/F | F
F → (E) | id | number

bottom-up: id*number-id ⇒ F*number-id ⇒ T*number-id ⇒ T*F-id ⇒ T-id ⇒ E-id ⇒ E-F ⇒ E-T ⇒ E

Stack (top is right)	**Input**	**Action**
$	id*number-id	shift
$ id	*number-id	reduce by F → id
$ F	*number-id	reduce by T → F
$ T	*number-id	shift
$ T *	number-id	shift
$ T * number	-id	reduce by F → number
$ T * F	-id	reduce by T → T * F
$ T	-id	reduce by E → T
$ E	-id	shift
$ E -	id	shift
$ E - id	$	reduce by F → id
$ E - F	$	reduce by T → F
$ E - T	$	reduce by E → E-T
$ E	$	accept

Copyright © 2019 Jonathan Cook

Bottom Up Parsing (TB Sec 4.5)

“Grammars used in compiling usually fall into the LR(1) class” (TB, p239)...but:
“An ambiguous grammar can never be LR” (TB, p239)

Two kinds of non-LR conflict in shift-reduce parsing:

1) **Shift/reduce conflict**: choose whether to shift or reduce
2) **Reduce/reduce conflict**: two+ reduction possibilities

These occur when **generating** the parser, not in parsing!
Both signal that the grammar is NOT LR(x)!

Parser generator could give up and signal an error...or, can simply choose how to resolve the conflict, and warn you!

Yacc: chooses shift in shift/reduce conflict, chooses topmost rule in yacc source file to reduce in reduce/reduce conflict

Important: the resulting parser is NOT ambigious!

Copyright © 2019 Jonathan Cook

Simple LR (SLR) Parsing (TB Sec 4.6)

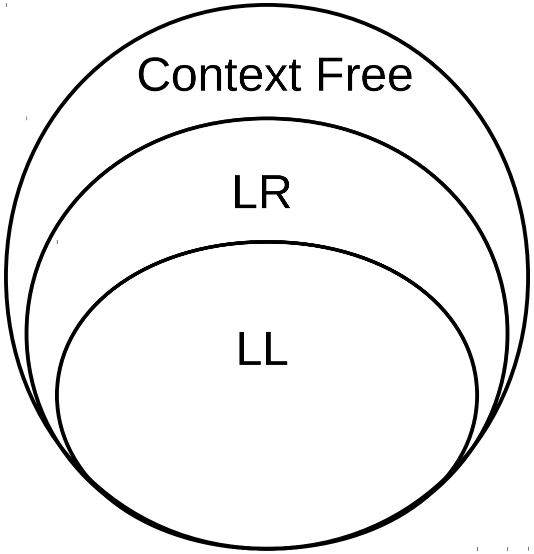
Sec 4.5 describes shift-reduce parsing, but did not detail any specific mechanism for identifying handles, generating a parser, or parsing

Sec 4.6 presents “the easiest method for constructing shift-reduce parsers” (TB, p241), called Simple LR, or SLR

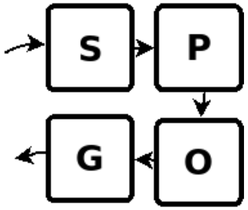
Sec 4.7 presents more complicated methods, one of which is LALR, which is what yacc uses; we won’t do this section

As in LL, basic goal is to construct a parsing table which encodes our rules and actions

“The class of grammars that can be parsed using LR methods is a proper superset of [those] that can be parsed with [.] LL methods” (TB, p242)



Copyright © 2019 Jonathan Cook



Simple LR (SLR) Parsing (TB Sec 4.6)

An LR(0) **item**, of a grammar G is a production rule of G with a **dot** at some position of the body of the rule; the dot represents how far the rule has been matched and what is hoped to be seen next on the input.

S → A B | ε

is shorthand for two rules; all dot positions are:

S → . A B
S → A . B
S → A B .
S → .

The S → ε rule is special: an empty rule generates only one dot position.

Copyright © 2019 Jonathan Cook

Simple LR (SLR) Parsing (TB Sec 4.6)

parser generator goal: construct “a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called an LR(0) automaton. In particular, each state of the LR(0) automaton represents a **set of items** in the **canonical LR(0) collection**.” (TB p243)

Two functions over set I of items: Closure and Goto

Closure(I):
1. add all items in I to Closure(I)
2. For item $A \rightarrow \alpha.B\beta$ in I and a production rule $B \rightarrow \gamma$, add item $B \rightarrow .\gamma$ to Closure(I) if not already in it. Repeat (2) until no new items added

Goto(I,X), where X is a grammar symbol:
1. Form set N of items $A \rightarrow \alpha X.\beta$ such that $A \rightarrow \alpha.X\beta$ is in I
2. Calculate and return Closure(N)

Copyright © 2019 Jonathan Cook

Simple LR (SLR) Parsing (TB Sec 4.6)

First: augment grammar with starting symbol S to have a new initial rule S’ → S ; we accept when just about to reduce this rule

Figure 4.33: Algorithm for computing canonical LR(0) item sets

```
C = {Closure(S’->.S)}
repeat:
   foreach (unprocessed I in C):
      foreach (grammar symbol X):
         J = Goto(I,X)
         if (J not empty and not in C):
            add J to C
      mark I as processed
until nothing new added to C
```

Our parsing automaton has a state for each set in the canonical LR(0) item sets, and the Goto() function defines the transitions (which occur on grammar symbols).

Copyright © 2019 Jonathan Cook

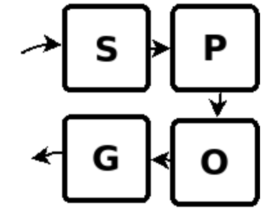
Simple LR (SLR) Parsing (TB Sec 4.6)

E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | id | number

(line #s below are just for reference)

1. Augment with E’ → E
2. Closure(E’→.E) is **S1** = {[E’→.E],[E→.E+T],[E→.E-T],[E→.T],
   [T→.T*F],[T→.T/F],[T→.F],[F→.(E)],[F→.id],[F→.number]}
3. Goto(S1,E) = Closure({[E’→E.],[E→E.+T],[E→E.-T]}) = **S2** =
   {[E’→E.],[E→E.+T],[E→E.-T]}
4. Goto(S1,T) = Closure({[E→T.],[T→T.*F],[T→T./F]}) = **S3** =
   {[E→T.],[T→T.*F],[T→T./F]}
5. Goto(S1,F) = Closure({[T→F.]}) = **S4** = {[T→F.]}
6. Goto(S1,[+,-,*,/,)]) = {}
7. Goto(S1,’(‘) = Closure({[F→(.E)]}) = **S5** = {[F→(.E)],[E→.E+T],
   [E→.E-T],[E→.T],[T→.T*F],[T→.T/F],[T→.F],[F→.(E)],
   [F→.id],[F→.number]}
8. Goto(S1,id) = Closure({[F→id.]}) = **S6** = {[F→id.]}
9. Goto(S1,number) = Closure({[F→number.]}) = **S7** = {[F→number.]}

Copyright © 2019 Jonathan Cook



Simple LR (SLR) Parsing (TB Sec 4.6)

E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | id | number

1. S2 = {[E'→E.],[E→E.+T],[E→E.-T]}
2. Goto(S2,[E,T,F,*,/,(,),id,number]) = {}
3. Goto(S2,+) = Closure({[E→E+.T]}) = **S8** = {[E→E+.T],[T→.T*F],
   [T→.T/F],[T→.F],[F→.(E)],[F→.id],[F→.number]}
4. Goto(S2,-) = Closure({[E→E-.T]}) = **S9** = {[E→E-.T],[T→.T*F],
   [T→.T/F],[T→.F],[F→.(E)],[F→.id],[F→.number]}

5. S3 = {[E→T.],[T→T.*F],[T→T./F]}
6. Goto(S3,[E,T,F,+,-,(,),id,number]) = {}
7. Goto(S3,*) = Closure({[T→T*.F]}) = **S10** = {[T→T*.F],[F→.(E)],
   [F→.id],[F→.number]}
8. Goto(S3,/) = Closure({[T→T/.F]}) = **S11** = {[T→T/.F],[F→.(E)],
   [F→.id],[F→.number]}

Copyright © 2019 Jonathan Cook

Simple LR (SLR) Parsing (TB Sec 4.6)

E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | id | number

1. S4 = {[T→F.]}
2. Goto(S4,[all]) = {}

3. S5 = {[F→(.E)],[E→.E+T],[E→.E-T],[E→.T],[T→.T*F],[T→.T/F],
   [T→.F],[F→.(E)],[F→.id],[F→.number]}
4. Goto(S5,E) = Closure({[F→(E.)],[E→E.+T],[E→E.-T]}) = **S12** =
   {[F→(E.)],[E→E.+T],[E→E.-T]}
5. Goto(S5,T) = Closure({[E→T.],[T→T.*F],[T→T./F]}) = **S13** =
   {[E→T.],[T→T.*F],[T→T./F]}}
6. Goto(S5,F) = Closure({[T→F.]}) = **S4**
7. Goto(S5,'(') = **S5**   Goto(S5,id) = **S6**   Goto(S5,num) = **S7**

8. S6 = {[F→id.]}, Goto(S6,all) = {}
9. S7 = {[F→number.]}, Goto(S7,all) = {}

Copyright © 2019 Jonathan Cook

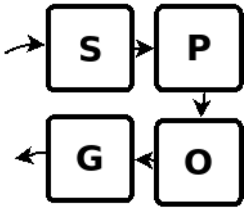
Simple LR (SLR) Parsing (TB Sec 4.6)

E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | id | number

1. S8 = {[E→E+.T],[T→.T*F],[T→.T/F],[T→.F],[F→.(E)],[F→.id], [F→.number]}
2. Goto(S8,T) = Closure({[E→E+T.],[T→T.*F],[T→T./F]}) = **S14** = {[E→E+T.],[T→T.*F],[T→T./F]}
3. Goto(S8,F) = Closure({[T→F.]}) = **S4** = {[T→F.]}
4. Goto(S8,’(‘) = Closure({[F→(.E)]}) = **S5**
5. Goto(S8,id) = **S6**   Goto(S8,number) = **S7**
6. Goto(S8,[E,+,-,*,/,)]) = {}

7. S9 = {[E→E-.T],[T→.T*F], [T→.T/F],[T→.F],[F→.(E)],[F→.id], [F→.number]}
8. Goto(S9,T) = Closure({[E→E-T.],[T→T.*F],[T→T./F]}) = **S15** = {[E→E-T.],[T→T.*F],[T→T./F]}
9. Goto(S9,F) = **S4**   Goto(S9,’(‘) = **S5**
10. Goto(S9,id) = **S6** Goto(S9,number) = **S7**
11. Goto(S9,[E,+,-,*,/,)]) = {}

Copyright © 2019 Jonathan Cook



Simple LR (SLR) Parsing (TB Sec 4.6)

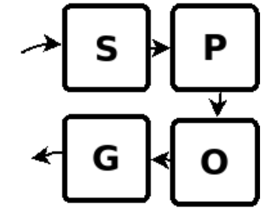
E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | id | number

1. S10 = {[T→T*.F],[F→.(E)],[F→.id],[F→.number]}
2. Goto(S10,F) = Closure({[T→T*F.]}) = **S16** = {[T→T*F.]}
3. Goto(S10,’(‘) = **S5** Goto(S10,id) = **S6** Goto(S10,number) = **S7**
4. Goto(S10,[E,T,+,-,*,/,)]) = {}

5. S11 = {[T→T/.F],[F→.(E)],[F→.id],[F→.number]}
6. Goto(S11,F) = Closure({[T→T/F.]}) = **S17** = {[T→T/F.]}
7. Goto(S11,’(‘) = **S5** Goto(S11,id) = **S6** Goto(S11,number) = **S7**
8. Goto(S11,[E,T,+,-,*,/,)]) = {}

9. S12 = {[F→(E.)],[E→E.+T],[E→E.-T]}
10. Goto(S12,+) = **S8** Goto(S12,-) = **S9** Goto(S12,else) = {}
11. Goto(S12,’)’) = Closure({[F→(E).]}) = {[F→(E).]} = **S18**

Copyright © 2019 Jonathan Cook



Simple LR (SLR) Parsing (TB Sec 4.6)

E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | id | number

1. S13 = {[E→T.],[T→T.*F],[T→T./F]}}
2. Goto(S13,*) = **S10**   Goto(S13,/) = **S11** Goto(S13,else) = {}

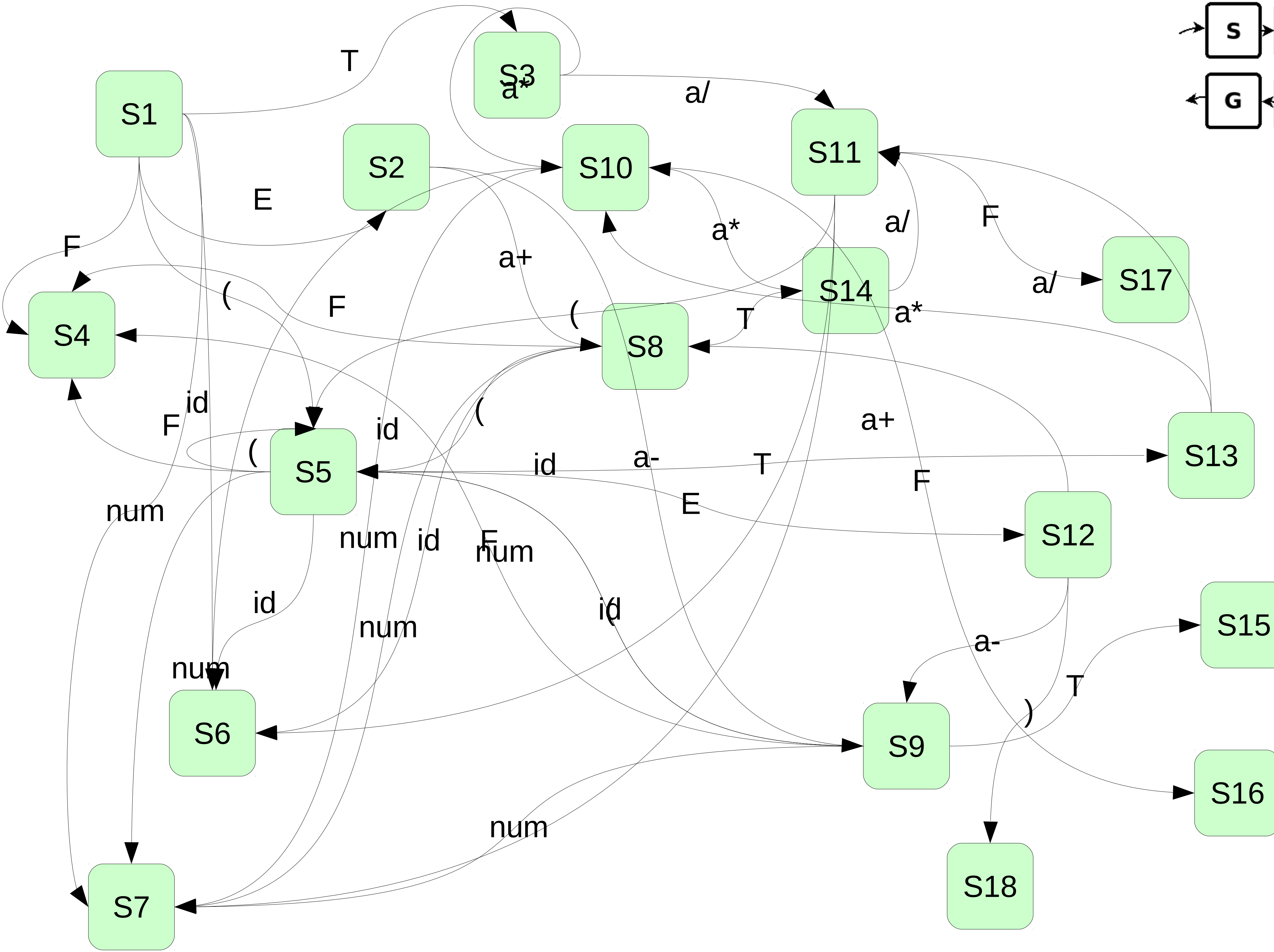
3. S14 = {[E→E+T.],[T→T.*F],[T→T./F]}
4. Goto(S14,*) = **S10**   Goto(S14,/) = **S11** Goto(S14,else) = {}

5. S15 = {[E→E-T.],[T→T.*F],[T→T./F]}
6. Goto(S15,*) = **S10**   Goto(S14,/) = **S11** Goto(S14,else) = {}

7. S16 = {[T→T*F.]} Goto(S16,all) = {}
8. S17 = {[T→T/F.]} Goto(S17,all) = {}
9. S18 = {[F→(E).]} Goto(S18,all) = {}

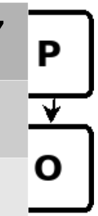
We’re done!, Now draw it!

Copyright © 2019 Jonathan Cook

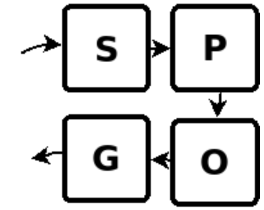


Copyright © 2019 Jonathan Cook

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17
S1		E	T	F	(	id	nm										
S2								+	-								
S3										*	/						
S4																	
S5				F	(	id	nm					E	T				
S6																	
S7																	
S8				F	(	id	nm							T			
S9				F	(	id	nm								T		
S10					(	id	nm									F	
S11					(	id	nm										F
S12								+	-								
S13										*	/						
S14										*	/						
S15										*	/						
S16																	
S17																	
S18												)					



Copyright © 2019 Jonathan Cook



Creating the Parsing Table(s) (Alg 4.46, p253)

Two tables: Action and GoTo (but we can put it in one)

For each Item set I:

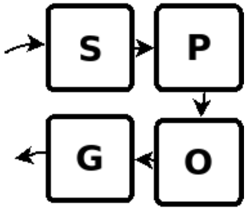
1. If $[A \rightarrow \alpha.a\beta]$ is in Ii and Goto(Ii,a)=Ij, then Action[i,a] = “shift j”
2. if $[A \rightarrow \alpha.]$ is in Ii, then Action[i,a] = “reduce $A \rightarrow \alpha$” for all a in FOLLOW(A)
3. if $[S' \rightarrow S.]$ in Ii then Action[i,\$] = “accept”
4. All other Action table entries are “error”

Goto table:

5. If Goto(Ii,A) = Ij for nonterminal A, then GoTo(i,A) = j

6. Initial state is from item set containing $[S' \rightarrow .S]$

Copyright © 2019 Jonathan Cook



Creating the Parsing Table(s) (Alg 4.46, p253)

Follow(E,T,F) = {+,-,*,/,),$}

S1: [1,(] = shift 5, [1,id] = shift 6, [1,num] = shift 7
Goto(1,E) = 2   Goto(1,T) = 3   Goto(1,F) = 4

S2: [2,+] = shift 8, [2,-] = shift 9; [2,$] = accept

S3: [3,*] = shift 10, [3,/] = shift 11, [3,{+,-,*,/,),$}] = reduce E $\rightarrow$ T (E3)

S4: [4,{+,-,*,/,),$}] = reduce T $\rightarrow$ F (T3)

S5: [5,(] = shift 5, [5,id] = shift 6, [5,num] = shift 7
Goto(5,E) = 12, Goto(5,T) = 13   Goto(5,F) = 4

S6: [6,{+,-,*,/,),$}] = reduce F $\rightarrow$ id (F2)

S7: [7,{+,-,*,/,),$}] = reduce F $\rightarrow$ num (F3)

S8: [8,(] = shift 5, [8,id] = shift 6, [8,num] = shift 7,   Goto(8,F)=4, Goto(8,T)=14

S9: [9,(] = shift 5, [9,id] = shift 6, [9,num] = shift 7,   Goto(9,F)=4, Goto(9,T)=15

Copyright © 2019 Jonathan Cook

Creating the Parsing Table(s) (Alg 4.46, p253)

S10: [10,(] = shift 5, [10,id] = shift 6, [10,num] = shift 7,   goto(10,F)=16

S11: [11,(] = shift 5, [11,id] = shift 6, [11,num] = shift 7,   goto(11,F)=17

S12: [12,+] = shift 8, [12,-] = shift 9

S13: [13,*] = shift 10, [13,/] = shift 11   [13,{+,-,*,/,),$}] = reduce E→T   (E3)

S14: [14,*] = shift 10, [14,/] = shift 11   [14,{+,-,*,/,),$}] = reduce E→E+T (E1)

S15: [15,*] = shift 10, [15,/] = shift 11   [15,{+,-,*,/,),$}] = reduce E→E-T   (E2)

S16: [16,{+,-,*,/,),$}] = reduce T→T*F   (T1)

S17: [17,{+,-,*,/,),$}] = reduce T→T/F   (T2)

S18: [18,{+,-,*,/,),$}] = reduce F→(E)   (F1)

Copyright © 2019 Jonathan Cook

	E	T	F	id	nm	+	-	*	/	(	)	$
S1	G 2	G 3	G 4	S 6	S 7					S 5		
S2						S 8	S 9					accept
S3						R E3	R E3	S 10	S 11		R E3	R E3
S4						R T3	R T3	R T3	R T3		R T3	R T3
S5	G 12	G 13	G 4	S 6	S 7							
S6						R F2	R F2	R F2	R F2		R F2	R F2
S7						R F3	R F3	R F3	R F3		R F3	R F3
S8		G 14	G 4	S 6	S 7					S 5		
S9		G 15	G 4	S 6	S 7					S 5		
S10			G 16	S 6	S 7					S 5		
S11			G 17	S 6	S 7					S 5		
S12						S 8	S 9					
S13						R E3	R E3	S 10	S 11		R E3	R E3
S14						R E1	R E1	S 10	S 11		R E1	R E1
S15						R E2	R E2	S 10	S 11		R E2	R E2
S16						R T1	R T1	R T1	R T1		R T1	R T1
S17						R T2	R T2	R T2	R T2		R T2	R T2
S18						R F1	R F1	R F1	R F1		R F1	R F1

Copyright © 2019 Jonathan Cook

Table-Driven Parsing Algorithm (Fig 4.36, p251)

- NOTE: stack is a stack of states, not of grammar symbols!

```
push starting state onto stack
a ← first symbol of w$
while (1) {
   s ← top state of stack (no pop)
   if (Action[s,a] = shift t) {
      push t onto stack
      a ← next symbol of w$
   } else if (Action[s,a] = reduce A→β) {
      pop |β| symbols off the stack
      t ← top state of stack
      push Goto(t,A) onto stack
      output production A→β
   } else if (Action[s,a] = accept) {
      break;
   } else {
      print syntax error and try to recover
   }
}
print success; output rules are RM derivation
```

Notes:
- states represent symbols *in context*
- reduction still happens when a *handle* is seen
- reduction still consumes N stack items (N == size of RHS of rule)

Copyright © 2019 Jonathan Cook

Bottom Up Parsing Using above Table

string: “i*42-j” == id * number - id

Stack (top is right)	Input	Action
1	id*number-id$	shift 6
1 6	*number-id$	reduce by F → id, goto 4
1 4	*number-id$	reduce by T → F, goto 3
1 3	*number-id$	shift 10
1 3 10	number-id$	shift 7
1 3 10 7	-id$	reduce by F → number, goto 16
1 3 10 16	-id$	reduce by T → T * F, goto 3
1 3	-id$	reduce by E → T, goto 2
1 2	-id$	shift 9
1 2 9	id$	shift 6
1 2 9 6	$	reduce by F → id, goto 4
1 2 9 4	$	reduce by T → F, goto 15
1 2 9 15	$	reduce by E → E-T, goto 2
1 2	$	accept!

Copyright © 2019 Jonathan Cook

Bottom Up Parsing: Example #2

Grammar:

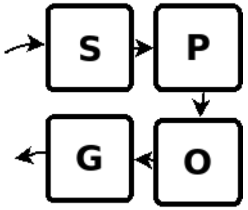
S → A S | empty   FOLLOW(S) = $
A → a B   FOLLOW(A) = a,$
B → b B | b   FOLLOW(B) = a,$
augment with: S' → S

Item sets:

S1 = Closure(S'→.S) = {S'→.S, S→.AS, S→., A→.aB}
Goto(S1,S) = Closure(S'→S.) = S2 = {S'→S.}
Goto(S1,A) = Closure(S→A.S) = S3 = {S→A.S, S→.AS, S→., A→.aB}
Goto(S1,a) = Closure(A→a.B) = S4 = {A→a.B, B→.bB, B→.b}
Goto(S3,S) = Closure(S→AS.) = S5 = {S→AS.}
Goto(S3,A) = Closure(S→A.S) = S3
Goto(S3,a) = Closure(A→a.B) = S4
Goto(S4,B) = Closure(A→aB.) = S7 = {A→aB.}
Goto(S4,b) = Closure({B→b.B, B→b.}) = S6 = {B→b.B, B→b., B→.bB, B→.b}
Goto(S6,B) = Closure({B→bB.}) = S8 = {B→bB.}
Goto(S6,b) = Closure(B→b.B, B→b.) = S6

all other goto's are empty; S2 is accept

Copyright © 2019 Jonathan Cook



	S1	S2	S3	S4	S5	S6	S7	S8
S1		S	A	a				
S2								
S3			A	a	S			
S4						b	B	
S5								
S6						b		B
S7								
S8								

Copyright © 2019 Jonathan Cook

	S	A	B	a	b	$
S1	goto 2	goto 3		shift 4		reduce S→e
S2						accept
S3	goto 5	goto 3		shift 4		reduce S→e
S4			goto 7		shift 6	
S5						reduce S→AS
S6			goto 8	reduce B→b	shift 6	reduce B→b
S7				reduce A→aB		reduce A→aB
S8				reduce B→bB		reduce B→bB

Copyright © 2019 Jonathan Cook

Bottom Up Parsing: Example #2

string: “abbab”

Stack (top is right)	Input	Action
1	abbab$	shift 4
1 4	bbab$	shift 6
1 4 6	bab$	shift 6
1 4 6 6	ab$	reduce B→b, goto 8
1 4 6 8	ab$	reduce B→bB, goto 7
1 4 7	ab$	reduce A→aB, goto 3
1 3	ab$	shift 4
1 3 4	b$	shift 6
1 3 4 6	$	reduce B→b, goto 7
1 3 4 7	$	reduce A→aB, goto 3
1 3 3	$	reduce S→e, goto 5
1 3 3 5	$	reduce S→AS, goto 5
1 3 5	$	reduce S→AS, goto 2
1 2	$	accept!

Copyright © 2019 Jonathan Cook

More Complicated LR Parsing (TB 4.7)

4.6 was just “simple LR”, SLR

4.7 introduces more complicated (and powerful) LR parsing

LR, or “canonical” LR: idea, add a terminal symbol to the definition of SLR items; this terminal will constrain the reduce actions to choose which rule to reduce based on the terminal (SLR applies the same reduce to all following terminals). This LR(1) method ends up having *many* items!

LALR, or “lookahead LR”: LR(1) generates many item sets with common “cores”; these can be merged (carefully)

LALR produces a number of item sets on the order of SLR, while LR(1) can be an order of magnitude bigger

Copyright © 2019 Jonathan Cook