Parsers

How do we combine tokens?

- Combine tokens ("words" in a language) to form programs ("sentences" in a language)
- Not all combinations of tokens are correct programs (not all sentences are grammatically correct)
- How do we define this?

Producing sentences

- Here are some possible rules for simplified English:
  - All sentences have a noun phrase, then a verb, then a noun phrase (a subject, a verb, an object)
  - Noun phrases are an article ("a" or "the"), an adjective ("black" or "big") and a noun ("cat" or "dog")
  - Verbs can be "eats" or "scratches"
- Sentences we can create:
  - "a black cat bites the big dog." "the big dog eats the black cat."
- Sentences we can't:
  - "cat scratches black dog." "dog the cat bites black."

More formally

S[entence]	→ P V P
[noun]P[hrase]	→ R A N
[a]R[ticle]	→ a | the
A[djective]	→ big | black
N[oun]	→ cat | dog
V[erb]	→ bites | scratches

Generating strings

S → A B $

A → A a

A → a

B → B b

B → b

- Productions tell us how to rewrite a non-terminal into a different set of symbols
  - Can rewrite non-terminals until we generate the string we want
- A parser's job: do this in reverse!
  - Figure out how a string was produced

To derive the string "a a b b b" we can do the following rewrites:

S ⇒ A B $ ⇒ A a B $ ⇒ a a B $ ⇒ a a B b $ ⇒ a a B b b $ ⇒ a a b b b $

Generalize

- Grammar G = (V_t, V_n, S, P)
  - V_t is the set of *terminals*
  - V_n is the set of *non-terminals*
  - S is the *start symbol*
  - P is the set of *productions*
    - Each production takes the form: $V_n \to \lambda \mid (V_n \mid V_t)+$
    - Grammar is *context-free* (why?)
- A simple grammar:

  G = ({a, b}, {S, A, B}, {S → A B $, A → A a, A → a, B → B b, B → b}, S)

Terminology

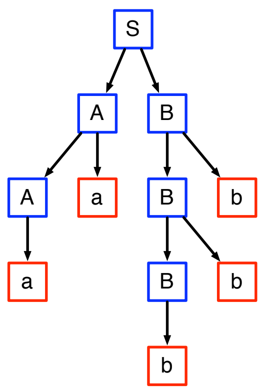
- V is the *vocabulary* of a grammar, consisting of terminal (V_t) and non-terminal (V_n) symbols
- For our sample grammar
  - V_n = {S, A, B}
    - Non-terminals are symbols on the LHS of a production
    - Non-terminals are constructs in the language that are recognized during parsing
  - V_t = {a, b}
    - Terminals are the tokens recognized by the scanner
    - They correspond to symbols in the text of the program

Terminology

- *Strings* are composed of symbols
  - A A a a B b b A a is a string
  - We will use Greek letters to represent strings composed of both terminals and non-terminals
- *L(G)* is the language produced by the grammar *G*
  - All strings consisting of only terminals that can be produced by G
  - In our example, L(G) = a+b+$
  - All regular expressions can be expressed as grammars for context-free languages, but not vice-versa
    - Consider: $a^i b^i$ $ (what is the grammar for this?)

Parse trees

- Tree which shows how a string was produced by a language
  - Interior nodes of tree: non-terminals
    - Children: the terminals and non-terminals generated by applying a production rule
  - Leaf nodes: terminals



Leftmost derivation

- Rewriting of a given string starts with the leftmost symbol
- Exercise: do a leftmost derivation of the input program

  F(V + V)

  using the following grammar:

E	→	Prefix (E)
E	→	V Tail
Prefix	→	F
Prefix	→	λ
Tail	→	+ E
Tail	→	λ

- What does the parse tree look like?

Rightmost derivation

- Rewrite using the rightmost non-terminal, instead of the left
- What is the rightmost derivation of this string?

  F(V + V)

E	→	Prefix (E)
E	→	V Tail
Prefix	→	F
Prefix	→	λ
Tail	→	+ E
Tail	→	λ

Simple conversions

A → B | C ⟶ A → B
A → C

D → E {F} ⟶ D → E Ftail
Ftail → F Ftail
Ftail → λ

Top-down vs. Bottom-up parsers

- Top-down parsers use left-most derivation
- Bottom-up parsers use right-looking parse
- Notation:
  - LL(1): Leftmost derivation with 1 symbol lookahead
  - LL(k): Leftmost derivation with k symbols lookahead
  - LR(1): Right-looking derivation with 1 symbol lookahead

Another simple grammar

PROGRAM → begin STMTLIST $

STMTLIST → STMT ; STMTLIST

STMTLIST → end

STMT → id

STMT → if (id) STMTLIST

- A sentence in the grammar:

  begin if (id) if (id) id ; end; end; end; $

- What are the terminals and non-terminals of this grammar?

Parsing this grammar

PROGRAM → begin STMTLIST $

STMTLIST → STMT ; STMTLIST

STMTLIST → end

STMT → id

STMT → if (id) STMTLIST

- Note
  - To parse STMT in STMTLIST → STMT; STMTLIST, it is necessary to choose between either STMT → id or STMT → if ...
  - Choose the production to parse by finding out if next token is if or id
    - i.e., which production the next input token matches
    - This is the *first* set of the production

Another example

S → A B $

A → x a A

A → y a A

A → λ

B → b

- Consider S ⇒ A B $ ⇒ x a A B $ ⇒ x a B $ ⇒ x a b $
- When parsing x a b $ we know from the goal production we need to match an A. The next token is x, so we apply A → x a A
- The parser matches x, matches a and now needs to parse A again
- How do we know which A to use? We need to use A → λ
  - When matching the right hand side of A → λ, the next token comes from a non-terminal that follows A (i.e., it must be b)
  - Tokens that can follow A are called the *follow* set of A

First and follow sets

- $\text{First}(\alpha) = \{a \in V_t \mid \alpha \Rightarrow^* a\beta\} \cup \{\lambda \mid \text{if } \alpha \Rightarrow^* \lambda\}$
- $\text{Follow}(A) = \{a \in V_t \mid S \Rightarrow^+ \ldots Aa \ldots\} \cup \{\$ \mid \text{if } S \Rightarrow^+ \ldots A\ \$\}$

S:	start symbol
a:	a terminal symbol
A:	a non-terminal symbol
α,β:	a string composed of terminals and non-terminals (typically, α is the RHS of a production

⇒:	derived in 1 step
$\Rightarrow^*$:	derived in 0 or more steps
$\Rightarrow^+$:	derived in 1 or more steps

First and follow sets

- First(α): the set of terminals that begin all strings that can be derived from α
  - First(A) = {x, y}
  - First(xaA) = {x}
  - First (AB) = {x, y, b}
- Follow(A): the set of terminals that can appear immediately after A in some partial derivation
  - Follow(A) = {b}

S → A B $

A → x a A

A → y a A

A → λ

B → b

Computing first sets

- Terminal: First(a) = {a}
- Non-terminal: First(A)
  - Look at all productions for A

    $A \rightarrow X_1X_2 \ldots X_k$
  - First(A) $\supseteq$ (First(X_1) - λ)
  - If $\lambda \in$ First(X_1), First(A) $\supseteq$ (First(X_2) - λ)
  - If λ is in First(X_i) for all i, then $\lambda \in$ First(A)
- Computing First(α): similar procedure to computing First(A)

Exercise

- What are the first sets for all the non-terminals in following grammar:

$S \rightarrow A\ B\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow \lambda$

$B \rightarrow b$

$B \rightarrow A$

Computing follow sets

- Follow(S) = {}
- To compute Follow(A):
  - Find productions which have A on rhs. Three rules:
    1. $X \rightarrow \alpha\ A\ \beta$: Follow(A) $\supseteq$ (First(β) - λ)
    2. $X \rightarrow \alpha\ A\ \beta$: If $\lambda \in$ First(β), Follow(A) $\supseteq$ Follow(X)
    3. $X \rightarrow \alpha\ A$: Follow(A) $\supseteq$ Follow(X)
- Note: Follow(X) never has λ in it.

Exercise

- What are the follow sets for

$S \rightarrow A\ B\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow \lambda$

$B \rightarrow b$

$B \rightarrow A$

Towards parser generators

- Key problem: as we read the source program, we need to decide what productions to use
- Step 1: find the tokens that can tell which production P (of the form $A \rightarrow X_1X_2 \ldots X_m$) applies

$$\text{Predict}(P) =$$

$$\begin{cases} \text{First}(\ \ _1 \quad\ _m) & \text{if } \lambda \notin \text{First}(\ \ _1 \quad\ _m) \\ (\text{First}(\ \ _1 \quad\ _m) - \lambda) \cup \text{Follow}(\ \ ) & \text{otherwise} \end{cases}$$

- If next token is in Predict(P), then we should choose this production

Parse tables

- Step 2: build a parse table
  - Given some non-terminal V_n (the non-terminal we are currently processing) and a terminal V_t (the lookahead symbol), the parse table tells us which production P to use (or that we have an error
  - More formally:

    $T: V_n \times V_t \rightarrow P \cup \{Error\}$

Building the parse table

- Start: T[A][t] = //initialize all fields to "error"

  foreach A:

  foreach P with A on its lhs:

  foreach t in Predict(P):

  T[A][t] = P

- Exercise: build parse table for our toy grammar

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

Stack-based parser for LL(1)

- Given the parse table, we can use a simple algorithm to parse programs
- Basic algorithm:
  1. Push the RHS of a production onto the stack
  2. Pop a symbol, if it is a terminal, match it
  3. If it is a non-terminal, take its production according to the parse table and go to 1
- Note: always start with start state

An example

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

- How would a stack-based parser parse:

  x a y a b

Parse stack	Remaining input	Parser action
S	x a y a b $	predict 1
A B $	x a y a b $	predict 2
x a A B $	x a y a b $	match(x)
a A B $	a y a b $	match(a)
A B $	y a b $	predict 3
y a A B $	y a b $	match(y)
a A B $	a b $	match(a)
A B $	b $	predict 4
B $	b $	predict 5
b $	b $	match(b)
$	$	Done!

LL(k) parsers

- Can use similar techniques for LL(k) parsers
- Use more than one symbol of look-ahead to distinguish productions
- Why might this be bad?

Dealing with semantic actions

- When a construct (corresponding to a production in a grammar) is recognized, a typical parser will invoke a semantic action
  - In a compiler, this action generates an intermediate representation of the program construct
  - In an interpreter, this action might be to perform the action specified by the construct. Thus, if *a+b* is recognized, the value of *a* and *b* would be added and placed in a temporary variable

Dealing with semantic actions

- We can annotate a grammar with *action symbols*
  - Tell the parser to invoke a semantic action routine
- Can simply push action symbols onto stack as well
- When popped, the semantic action routine is called
  - Routine manipulates *semantic records* on a stack
  - Can generate new records (e.g., to store variable info)
  - Can generate code using existing records
- Example: semantic actions for x = a + 3

```
statement ::= ID #id = expr #assign
expr ::= term + term #addop
term ::= ID #id | LITERAL #num
```

Non-LL(1) grammars

- Not all grammars are LL(1)!
- Consider

  <stmt> → if <expr> then <stmt list> endif

  <stmt> → if <expr> then <stmt list> else <stmt list> endif
- This is not LL(1) (why?)
- We can turn this in to

  <stmt> → if <expr> then <stmt list> <if suffix>

  <if suffix> → endif

  <if suffix> → else <stmt list> endif

Left recursion

- *Left recursion* is a problem for LL(1) parsers
  - LHS is also the first symbol of the RHS
- Consider:

  E → E + T
- What would happen with the stack-based algorithm?

Removing left recursion

E → E + T
E → T

→

E → E1 Etail
E1 → T
Etail → + T Etail
Etail → λ

Are all grammars LL(k)?

- No! Consider the following grammar:

  S → E
  E → (E + E)
  E → (E – E)
  E → x
- When parsing E, how do we know whether to use rule 2 or 3?
  - Potentially unbounded number of characters before the distinguishing '+' or '–' is found
  - No amount of lookahead will help!

In real languages?

- Consider the if-then-else problem
- `if x then y else z`
- Problem: else is optional
- `if a then if b then c else d`
  - Which `if` does the `else` belong to?
- This is analogous to a "bracket language": $[^i\,]^j$ ($i \geq j$)

  S → [S C
  S → λ
  C →]
  C → λ

  [[ ] can be parsed: SSλC or SSCλ
  (it's ambiguous!)

Solving the if-then-else problem

- The ambiguity exists at the language level. To fix, we need to define the semantics properly
  - "] matches nearest unmatched ["
  - This is the rule C uses for if-then-else
  - What if we try this?

  S → [S
  S → S1
  S1 → [S1]
  S1 → λ

  This grammar is still not LL(1) (or LL(k) for any k!)

Two possible fixes

- If there is an ambiguity, prioritize one production over another
  - e.g., if C is on the stack, always match "]" before matching "λ"

S → [S C
S → λ
C →]
C → λ

- Another option: change the language!
  - e.g., all if-statements need to be closed with an endif

S → if S E
S → other
E → else S endif
E → endif

Parsing if-then-else

- What if we don't want to change the language?
  - C does not require { } to delimit single-statement blocks
- To parse if-then-else, *we need to be able to look ahead at the entire rhs of a production* before deciding which production to use
  - In other words, we need to determine how many "]" to match before we start matching "["s
- *LR parsers* can do this!

LR Parsers

- Parser which does a Left-to-right, Right-most derivation
  - Rather than parse top-down, like LL parsers do, parse bottom-up, starting from leaves
- Basic idea: put tokens on a stack until an entire production is found
- Issues:
  - Recognizing the endpoint of a production
  - Finding the length of a production (RHS)
  - Finding the corresponding nonterminal (the LHS of the production)

LR Parsers

- Basic idea:
  - **shift** tokens onto the stack. At any step, keep the set of productions that could generate the read-in tokens
  - **reduce** the RHS of recognized productions to the corresponding non-terminal on the LHS of the production. Replace the RHS tokens on the stack with the LHS non-terminal.

Data structures

- At each state, given the next token,
  - A *goto table* defines the successor state
  - An *action table* defines whether to
    - *shift* – put the next state and token on the stack
    - *reduce* – an RHS is found; process the production
    - *terminate* – parsing is complete

Simple example

1. P → S
2. S → x ; S
3. S → e

		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Parsing using an LR(0) parser

- Basic idea: parser keeps track, simultaneously, of all possible productions that *could be matched* given what it's seen so far. When it sees a full production, match it.
- Maintain a *parse stack* that tells you what state you're in
  - Start in state 0
- In each state, look up in action table whether to:
  - *shift*: consume a token off the input; look for next state in goto table; push next state onto stack
  - *reduce*: match a production; pop off as many symbols from state stack as seen in production; look up where to go according to non-terminal we just matched; push next state onto stack
  - *accept*: terminate parse

Example

- Parse "x ; x ; e"

Step	Parse Stack	Remaining Input	Parser Action
1	0	x ; x ; e	Shift 1
2	0 1	; x ; e	Shift 2
3	0 1 2	x ; e	Shift 1
4	0 1 2 1	; e	Shift 2
5	0 1 2 1 2	e	Shift 3
6	0 1 2 1 2 3		Reduce 3 (goto 4)
7	0 1 2 1 2 4		Reduce 2 (goto 4)
8	0 1 2 4		Reduce 2 (goto 5)
9	0 5		Accept

LR(k) parsers

- LR(0) parsers
  - No lookahead
  - Predict which action to take by looking only at the symbols currently on the stack
- LR(k) parsers
  - Can look ahead *k* symbols
  - Most powerful class of deterministic bottom-up parsers
  - LR(1) and variants are the most common parsers

Terminology for LR parsers

- Configuration: a production augmented with a "•"

  $A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j$
- The "•" marks the point to which the production has been recognized. In this case, we have recognized $X_1 \ldots X_i$
- Configuration set: all the configurations that can apply at a given point during the parse:

  A → B • CD

  A → B • GH

  T → B • Z
- Idea: every configuration in a configuration set is a production that we could be in the process of matching

Configuration closure set

- Include all the configurations necessary to recognize the next symbol after the •
- For each configuration in set:
  - If next symbol is terminal, no new configuration added
  - If next symbol is non-terminal A, for each production of the form X → α, add configuration X → •α

S → E $
E → E + T | T
T → ID | (E)

closure0({S → • E $}) = {
  S → • E $
  E → • E + T
  E → • T
  T → • ID
  T → • (E)
}

Successor configuration set

- Starting with the initial configuration set

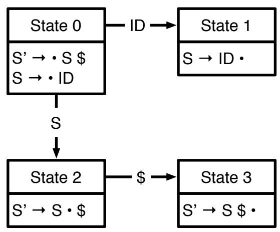
  s0 = closure0({S → • α $})

  an LR(0) parser will find the successor given the next symbol X
- X can be either a terminal (the next token from the scanner) or a non-terminal (the result of applying a reduction)
- Determining the successor s' = go_to0(s, X):
  - For each configuration in s of the form A → β • X γ add A → β X • γ to t
  - s' = closure0(t)

CFSM

- CFSM = Characteristic Finite State Machine
- Nodes are configuration sets (starting from s0)
- Arcs are go_to relationships

S' → S $
S → ID



Building the goto table

- We can just read this off from the CFSM

		Symbol		
		ID	$	S
State	0	1		2
	1			
	2		3	
	3			

Building the action table

- Given the configuration set s:
  - We shift if the next token matches a terminal after the • in some configuration

    A → α • a β ∈ s and a ∈ V_t, else error
  - We reduce production P if the • is at the end of a production

    B → α • ∈ s where production P is B → α
  - Extra actions:
    - shift if goto table transitions between states on a non-terminal
    - accept if we have matched the goal production

Action table

		Symbol		
		ID	$	S
State	0	S		S
	1	R2	R2	R2
	2		S	
	3	A	A	A

Conflicts in action table

- For LR(0) grammars, the action table entries are unique: from each state, can only shift or reduce
- But other grammars may have conflicts
  - Reduce/reduce conflicts: multiple reductions possible from the given configuration
  - Shift/reduce conflicts: we can either shift or reduce from the given configuration

Shift/reduce example

- Consider the following grammar:

  S → A y

  A → λ | x
- This leads to the following initial configuration set:

  S → • A y

  A → • x

  A → λ •
- Can shift or reduce here

Lookahead

- Can resolve reduce/reduce conflicts and shift/reduce conflicts by employing *lookahead*
  - Looking ahead one (or more) tokens allows us to determine whether to shift or reduce
  - (*cf* how we resolved ambiguity in LL(1) parsers by looking ahead one token)

Semantic actions

- Recall: in LL parsers, we could integrate the semantic actions with the parser
  - Why? Because the parser was *predictive*
- Why doesn't that work for LR parsers?
  - Don't know which production is matched until parser reduces
- For LR parsers, we put semantic actions at the end of productions
  - May have to rewrite grammar to support all necessary semantic actions

Parsers with lookahead

- Adding lookahead creates an LR(1) parser
  - Built using similar techniques as LR(0) parsers, but uses lookahead to distinguish states
  - LR(1) machines can be much larger than LR(0) machines, but resolve many shift/reduce and reduce/reduce conflicts
  - Other types of LR parsers are SLR(1) and LALR(1)
    - Differ in how they resolve ambiguities
    - yacc and bison produce LALR(1) parsers

LR(1) parsing

- Configurations in LR(1) look similar to LR(0), but they are extended to include a lookahead symbol

  $A \to X_1 \dots X_i \bullet X_{i+1} \dots X_j , l$ (where $l \in V_t \cup \lambda$)

- If two configurations differ only in their lookahead component, we combine them

  $A \to X_1 \dots X_i \bullet X_{i+1} \dots X_j , \{l_1 \dots l_m\}$

Building configuration sets

- To close a configuration

  $B \to \alpha \bullet A\ \beta, l$

- Add all configurations of the form $A \to \bullet\ \gamma, u$ where $u \in \mathrm{First}(\beta l)$
- Intuition: the lookahead symbol for any configuration is the terminal we expect to see *after the configuration has been matched*
  - The parse could apply the production for A, and the lookahead after we apply the production should match the next token that would be produced by B

Example

$S \to E\ \$$
$E \to E + T \mid T$
$T \to ID \mid (E)$

closure1({S → • E $, {λ}}) =
S → • E $, {λ}
E → • E + T, {$}
E → • T, {$}
T → • ID, {$}
T → • (E), {$}
E → • E + T, {+}
E → • T, {+}
T → • ID, {+}
T → • (E), {+}

Building goto and action tables

- The function goto1(configuration-set, symbol) is analogous to goto0(configuration-set, symbol) for LR(0)
  - Build goto table in the same way as for LR(0)
- Key difference: the action table.

  action[s][x] =

  - reduce when • is at end of configuration *and* x ∈ lookahead set of configuration

    $A \to \alpha \bullet, \{\ldots x \ldots\} \in s$

  - shift when • is before x

    $A \to \beta \bullet x \gamma \in s$

Example

- Consider the simple grammar:

  <program> → begin <stmts> end $

  <stmts> → SimpleStmt ; <stmts>

  <stmts> → begin <stmts> end ; <stmts>

  <stmts> → λ

Action and goto tables

	begin	end	;	SimpleStmt	$	<program>	<stmts>
0	S / 1						
1	S / 4	R4		S / 5			S / 2
2		S / 3					
3					A		
4	S / 4	R4		S / 5			S / 7
5			S / 6				
6	S / 4	R4		S / 5			S / 10
7		S / 8					
8			S / 9				
9	S / 4	R4		S / 6			S / 11
10		R2					
11		R3					

<program> → begin <stmts> end $

<stmts> → SimpleStmt ; <stmts>

<stmts> → begin <stmts> end ; <stmts>

<stmts> → λ

Example

- Parse: begin SimpleStmt ; SimpleStmt ; end $

Step	Parse Stack	Remaining Input	Parser Action
1	0	begin S ; S ; end $	Shift 1
2	0 1	S ; S ; end $	Shift 5
3	0 1 5	; S ; end $	Shift 6
4	0 1 5 6	S ; end $	Shift 5
5	0 1 5 6 5	; end $	Shift 6
6	0 1 5 6 5 6	end $	Reduce 4 (goto 10)
7	0 1 5 6 5 6 10	end $	Reduce 2 (goto 10)
8	0 1 5 6 10	end $	Reduce 2 (goto 2)
9	0 1 2	end $	Shift 3
10	0 1 2 3	$	Accept

Problems with LR(1) parsers

- LR(1) parsers are very powerful ...
  - But the table size is much larger than LR(0) — as much as a factor of $|V_t|$ (why?)
  - Example: Algol 60 (a simple language) includes several thousand states!
- Storage efficient representations of tables are an important issue

Solutions to the size problem

- Different parser schemes
  - SLR (simple LR): build an CFSM for a language, then add lookahead wherever necessary (*i.e.*, add lookahead to resolve shift/reduce conflicts)
    - What should the lookahead symbol be?
    - To decide whether to reduce using production A → α, use Follow(A)
  - LALR: merge LR states in certain cases (we won't discuss this)