COS320: Compiling Techniques

Zak Kincaid

March 7, 2019

Parsing II: LL parsing

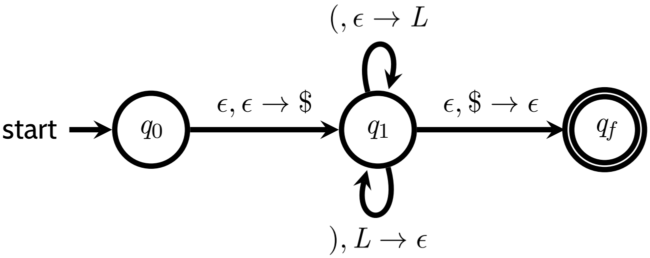
Recall: Context-free grammars

- A *context-free grammar* $G = (N, \Sigma, R, S)$ consists of:
  - N: a finite set of *non-terminal symbols*
  - Σ: a finite alphabet (or set of *terminal symbols*)
  - $R \subseteq N \times (N \cup \Sigma)^*$ a finite set of *rules* or *productions*
  - $S \in N$: the starting non-terminal.
- A *derivation* consists of a finite sequence of words $\gamma_1, \ldots, \gamma_n \in (N \cup \Sigma)^*$ such that $\gamma_1 = S$ and for each i, γ_{i+1} is obtained from γ_i by replacing a non-terminal symbol with the right-hand-side of one of its rules
- The set of all strings $w \in \Sigma^*$ such that G has a derivation of w is the *language* of G, written $\mathcal{L}(G)$.

Parsing

- Context-free grammars are *generative*: easy to find strings that belongs to $\mathcal{L}(G)$, not so easy determine whether a *given* string belongs to $\mathcal{L}(G)$
- *Pushdown automata* (PDA) are a kind of automata that recognize context-free languages

- Pushdown automaton recognizing <S> ::= <S><S> | (<S>) | ϵ:
    - *Stack alphabet*: $\$$ marks bottom of the stack, L marks unbalanced left paren

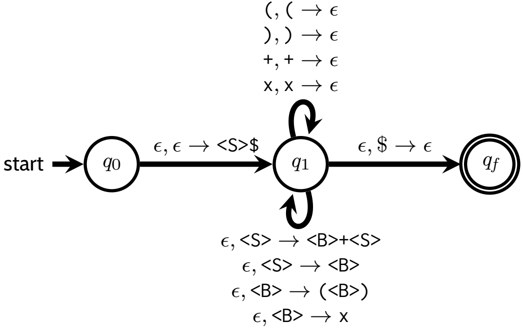


Top-down parsing

- Stack represents intermediate state of a derivation, minus the consumed part of the input string.
- Start with S on the stack
- Any time top of the stack is a non-terminal A, non-deterministically choose a rule $A ::= \gamma \in R$. Pop A off the stack, and push γ
- If the top of the stack is a terminal a, consume a from the input string and pop a off the stack
- Accept when stack is empty

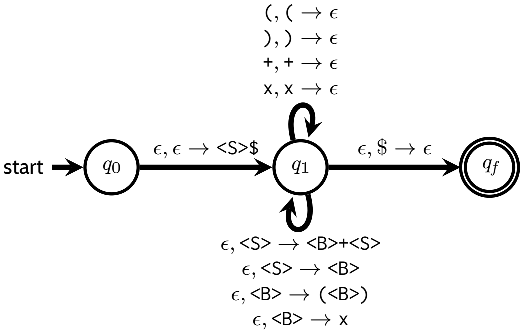
<S> ::= <B>+<S> | <B>

<B> ::= (<S>) | x



<S> ::= <B>+<S> | <B>

<B> ::= (<S>) | x



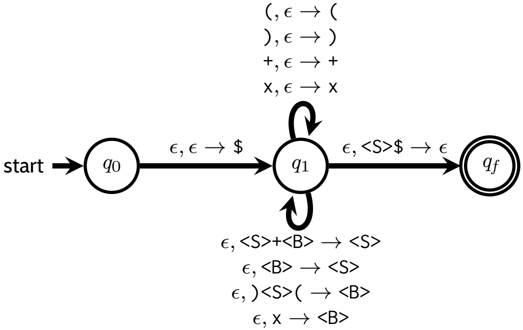
State	Stack	Input
q_0	ϵ	(x+x)+x
q_1	<S>$	(x+x)+x
q_1	<B>+<S>$	(x+x)+x
q_1	(<S>)+<S>$	(x+x)+x
q_1	<S>)+<S>$	x+x)+x
q_1	<B>+<S>)+<S>$	x+x)+x
q_1	x+<S>)+<S>$	x+x)+x
q_1	+<S>)+<S>$	+x)+x
q_1	<S>)+<S>$	x)+x
q_1	<B>)+<S>$	x)+x
q_1	x)+<S>$	x)+x
q_1	)+<S>$	)+x
q_1	+<S>$	+x
q_1	<S>$	x
q_1	<B>$	x
q_1	x$	x
q_1	$	ϵ
q_f	ϵ	ϵ

Bottom-up parsing

- Stack holds a word in $(N \cup \Sigma)^*$ from which it is possible to derive the part of the input string that has been consumed
- At any time, may read a letter from input string and push it on top of the stack
- At any time, may non-deterministically choose a rule $A ::= \gamma_1 ... \gamma_n$ and apply it in reverse: pop $\gamma_n ... \gamma_1$ off the top of the stack, and push A.
- Accept when stack just contains start non-terminal

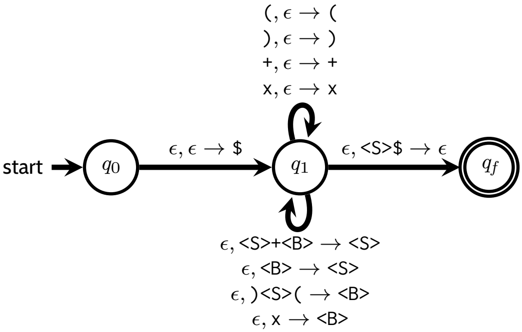
<S> ::= <B>+<S> | <B>

<B> ::= (<S>) | x



<S> ::= <B>+<S> | <B>

<B> ::= (<S>) | x



State	Stack	Input
q_0	ϵ	(x+x)+x
q_1	\$	(x+x)+x
q_1	(\$	x+x)+x
q_1	x(\$	+x)+x
q_1	<B>(\$	+x)+x
q_1	+<B>(\$	x)+x
q_1	x+<B>(\$	)+x
q_1	<B>+<B>(\$	)+x
q_1	<S>+<B>(\$	)+x
q_1	<S>(\$	)+x
q_1	)<S>(\$	+x
q_1	<B>\$	+x
q_1	+<B>\$	x
q_1	x+<B>\$	ϵ
q_1	<B>+<B>\$	ϵ
q_1	<S>+<B>\$	ϵ
q_1	<S>\$	ϵ
q_f	ϵ	ϵ

Parsing overview

- Basic problem with both top-down and bottom-up construction: *non-determinism*
  - Non-deterministic search is inefficient
    - E.g., consider <S> ::= <S>a | <S>b | ϵ. Top-down parser must "guess" the entire input string at the beginning (breadth-first backtracking search takes exponential time in length of input string, depth-first does not terminate).
  - Algorithms for parsing any context free grammar in cubic[1] time, based on dynamic programming (Earley, and Cocke-Younger-Kasami).

[1] Also sub-cubic galactic algorithms

Parsing overview

- Basic problem with both top-down and bottom-up construction: *non-determinism*
  - Non-deterministic search is inefficient
    - E.g., consider <S> ::= <S>a | <S>b | ϵ. Top-down parser must "guess" the entire input string at the beginning (breadth-first backtracking search takes exponential time in length of input string, depth-first does not terminate).
  - Algorithms for parsing any context free grammar in cubic[1] time, based on dynamic programming (Earley, and Cocke-Younger-Kasami).
- Parser generators use these same ideas, but restricted to cases where we can eliminate non-determinism.
- Possible for both top-down and bottom-up style
  - **Today:** *LL* (*L*eft-to-right, *L*eftmost derivation) parsers: top-down
    - Easy to understand & write by hand
  - **Next week:** *LR* (*L*eft-to-right, *R*ightmost derivation) parsers: bottom-up
    - More general, (variations) implemented in parser generators

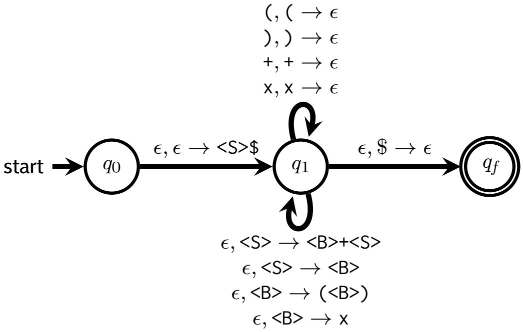
---

[1] Also sub-cubic galactic algorithms

LL parsing

<S> ::= <B>+<S> | <B>

<B> ::= (<S>) | x



- "Any time top of the stack is a non-terminal A, non-deterministically choose a production $A ::= \gamma \in R$. Pop A off the stack, and push γ"
  - Key problem: need to deterministically choose which production to use
  - Solution: Look at the next input symbol, but don't consume it (*lookahead*)
    - This is $LL(1)$ parsing. $LL(k)$ allows k lookahead tokens

- We say that a grammar is $LL(k)$ if we look ahead k symbols in a top-down parser, we know which rule we should apply.
  - Let $G = (N, \Sigma, R, S)$ be a grammar. G is $LL(k)$ iff: for any $S \Rightarrow^* \alpha A \beta$, for any word $w \in \Sigma^k$, if there is some $A ::= \gamma \in R$ such that $\gamma\beta \Rightarrow^* w\beta'$ (for some β'), then γ is unique.
- Not every context-free language has an $LL(k)$ grammar.
  - $\{a^i b^j : i = j \vee 2i = j\}$ is not $LL(k)$ for any k
- Which of the following are $LL(1)$ grammars?
  - <S> ::= a<S> | b<S> | ϵ

  - <S> ::= <S>a | <S>b | ϵ
  - <S> ::= <B>+<S> | <B>
    <B> ::= (<S>) | x

- We say that a grammar is $LL(k)$ if we look ahead k symbols in a top-down parser, we know which rule we should apply.
  - Let $G = (N, \Sigma, R, S)$ be a grammar. G is $LL(k)$ iff: for any $S \Rightarrow^* \alpha A \beta$, for any word $w \in \Sigma^k$, if there is some $A ::= \gamma \in R$ such that $\gamma\beta \Rightarrow^* w\beta'$ (for some β'), then γ is unique.
- Not every context-free language has an $LL(k)$ grammar.
  - $\{a^i b^j : i = j \vee 2i = j\}$ is not $LL(k)$ for any k
- Which of the following are $LL(1)$ grammars?
  - <S> ::= a<S> | b<S> | ϵ
    More generally, any grammar that results from our DFA→CFG conversion
  - <S> ::= <S>a | <S>b | ϵ
  - <S> ::= <B>+<S> | <B>
    <B> ::= (<S>) | x

Left-factoring

- The grammar

<S> ::= <B>+<S> | <B>

<B> ::= (<S>) | x

is not LL(1): (lookahead can't distinguish the two <S> rules

- However, there is an LL(1) grammar for the language

Left-factoring

- The grammar

<S> ::= <B>+<S> | <B>

<B> ::= (<S>) | x

is not LL(1): (lookahead can't distinguish the two <S> rules

- However, there is an LL(1) grammar for the language

<S> ::= <B><R>

<R> ::= +<S> | ϵ

<B> ::= (<S>) | x

- General strategy: factor out rules with common prefixes ("left factoring")

Eliminating left recursion

- A grammar is **left-recursive** if there is a non-terminal A such that $A \Rightarrow^{+} A\gamma$ (for some γ)
- Left-recursive grammars are not $LL(k)$ for any k
- Consider:

```
<S> ::= <S>+<B> | <B>
<B> ::= (<S>) | x
```

Can remove left recursion as follows:

```
 <S> ::= <B><S'>
<S'> ::= +<B><S'> | ϵ
 <B> ::= (<S>) | x
```

(Recognizes the same language, but parse trees are different!)

Mechanical construction of LL(1) parsers

- Fix a grammar $G = (N, \Sigma, R, S)$
- For any word $\gamma \in (N \cup \Sigma)^*$, define **first**$(\gamma) = \{a \in \Sigma : \gamma \Rightarrow^* aw\}$
- For any word $\gamma \in (N \cup \Sigma)^*$, say that γ is **nullable** if $\gamma \Rightarrow^* \epsilon$
- For any non-terminal A, define **follow**$(A) = \{a \in \Sigma : \exists \gamma, \gamma'. S \Rightarrow \gamma A a \gamma'\}$
- Transition table for G can be computed using **first**, **follow**, and **nullable**:
  1. For each non-terminal A and letter a, initialize $\Gamma(A, a)$ to $\emptyset$
  2. For each rule $A ::= \gamma$
     - Add γ to $\Gamma(A, a)$ for each $a \in$ **first**(γ)
     - If γ is nullable, add γ to $\Gamma(A, a)$ for each $a \in$ **follow**(A)

Mechanical construction of LL(1) parsers

- Fix a grammar $G = (N, \Sigma, R, S)$
- For any word $\gamma \in (N \cup \Sigma)^*$, define first$(\gamma) = \{a \in \Sigma : \gamma \Rightarrow^* aw\}$
- For any word $\gamma \in (N \cup \Sigma)^*$, say that γ is nullable if $\gamma \Rightarrow^* \epsilon$
- For any non-terminal A, define follow$(A) = \{a \in \Sigma : \exists \gamma, \gamma'. S \Rightarrow \gamma A a \gamma'\}$
- Transition table for G can be computed using **first**, **follow**, and **nullable**:
  1. For each non-terminal A and letter a, initialize $\Gamma(A, a)$ to $\emptyset$
  2. For each rule $A ::= \gamma$
     - Add γ to $\Gamma(A, a)$ for each $a \in$ **first**(γ)
     - If γ is nullable, add γ to $\Gamma(A, a)$ for each $a \in$ **follow**(A)
- G is $LL(1)$ iff $\Gamma(A, a)$ is empty or singleton for all A and a

Mechanical construction of LL(1) parsers

- Fix a grammar $G = (N, \Sigma, R, S)$
- For any word $\gamma \in (N \cup \Sigma)^*$, define first$(\gamma) = \{a \in \Sigma : \gamma \Rightarrow^* aw\}$
- For any word $\gamma \in (N \cup \Sigma)^*$, say that γ is nullable if $\gamma \Rightarrow^* \epsilon$
- For any non-terminal A, define follow$(A) = \{a \in \Sigma : \exists \gamma, \gamma'. S \Rightarrow \gamma A a \gamma'\}$
- Transition table for G can be computed using **first**, **follow**, and **nullable**:
  1. For each non-terminal A and letter a, initialize $\Gamma(A, a)$ to $\emptyset$
  2. For each rule $A ::= \gamma$
     - Add γ to $\Gamma(A, a)$ for each $a \in$ **first**(γ)
     - If γ is nullable, add γ to $\Gamma(A, a)$ for each $a \in$ **follow**(A)
- G is $LL(1)$ iff $\Gamma(A, a)$ is empty or singleton for all A and a
- Operation of the parser on a word w:
  - Start with stack <S>
  - While w not empty
    - If top of the stack is a terminal a and $w = aw'$, pop and set $w = w'$
    - If top of the stack is a non-terminal A and $w = aw'$, pop and push (singleton) $\Gamma(A, w)$ (or reject of $\Gamma(A, w)$ is empty)
  - Accept if stack is empty; reject otherwise.

Computing nullable

- **nullable** is the *smallest set* of non-terminals such that if there is some $A ::= \gamma_1 \ldots \gamma_n \in R$ with $\gamma_1, \ldots, \gamma_n \in$ **nullable** implies $A \in$ **nullable**
  - Fixpoint computation:
    - $\textbf{nullable}_0 = \emptyset$
    - $\textbf{nullable}_{i+1} = \{A : \exists \gamma_1, \ldots, \gamma_n \in \textbf{nullable}_i . A ::= \gamma_1 \ldots \gamma_n \in R\}$
    - $\textbf{nullable} = \bigcup_{i=0}^{\infty} \textbf{nullable}_i$

```
nullable ← ∅;
changed ← true;
while changed do
    changed ← false;
    for A := γ1…γn ∈ R do
        if A ∉ nullable ∧ γ1, …, γn ∈ nullable then
            nullable ← nullable ∪ {A};
            changed ← true;
```

- Fixpoint computations appear everywhere!
  - Later we will see how they are used in dataflow analysis

Computing first and follow

- **first** is the *smallest function*[2] such that
  - For each $a \in \Sigma$, **first**$(a) = \{a\}$
  - For each $A ::= \gamma_1 ... \gamma_i ... \gamma_n \in R$, with $\gamma_1, ..., \gamma_{i-1}$ nullable, **first**$(A) \supseteq$ **first**(γ_i)
- **follow** is the *smallest function* such that
  - For each $A ::= \gamma_1 ... \gamma_i ... \gamma_n \in R$, with $\gamma_{i+1}, ..., \gamma_n$ nullable, **follow**$(\gamma_i) \supseteq$ **follow**(A)
  - For each $A ::= \gamma_1 ... \gamma_i ... \gamma_j ... \gamma_n \in R$, with $\gamma_{i+1}, ..., \gamma_{j-1}$ nullable, **follow**$(\gamma_i) \supseteq$ **first**(A)
- Both can be computed using a fixpoint algorithm, like **nullable**

[2] Pointwise order: $f \leq g$ if for all x, $f(x) \leq g(x)$