COS320: Compiling Techniques

Zak Kincaid

March 3, 2020

- Reminder: HW2 due today
- HW3 on course webpage. Due March 31. **Start early!**
  - You will implement a compiler for a simple imperative programming language (Oat), targetting LLVMlite.
  - You may work individually or in pairs
- Midterm next Thursday
  - Covers material in lectures up to March 5th (this Thursday)
    - Interpreters, program transformation, X86, IRs, lexing, parsing
  - How to prepare
    - Start on HW3
    - Review slides
    - Review example code from lectures (try re-implementing!)
  - Review next Tuesday: come prepared with questions

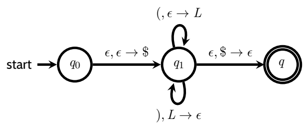
Parsing II: LL parsing

Recall: Context-free grammars

- · *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*)
  - $R \subseteq N \quad (N \cup \Sigma)^*$ a finite set of *rules* or *productions*
  - $S \in N$: the starting non-terminal.
- · *derivation* consists of a finite sequence of words $\gamma_1, ..., \gamma_n \in (N \cup \Sigma)$ 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

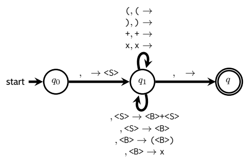
- Context-free grammars are *generative*: easy to find strings that belongs to $\mathcal{L}(G)$, not so easy determine whether a *given* string belongs to $\mathcal{L}(G)$
- *Pushdown automata* (PD·) are a kind of automata that recognize context-free languages
- Pushdown automaton recognizing <S> ::= <S><S> | (<S>) | ϵ:
  - *Stack alphabet*: \$ marks bottom of the stack, L marks unbalanced left paren



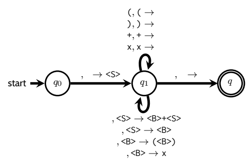
Top-down parsing

- Stack represents intermediate state of a derivation, minus the consumed part of the input string.
- Start with S on the stack
- · ny time top of the stack is a non-terminal , non-deterministically choose a rule $::= \gamma \in R.$ Pop 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
- · ccept when stack is empty

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



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

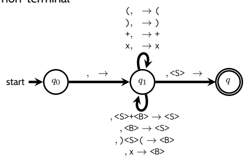


State	Stack	·nput
q_0		(x+x)+x
q_1	<S>	(x+x)+x
q_1	<B>+<S>	(x+x)+x
q_1	(<S>)+<S>	(x+x)+x
q_1	<S>)+<S>	x+x)+x
q_1	<B>+<S>)+<S>	x+x)+x
q_1	x+<S>)+<S>	x+x)+x
q_1	+<S>)+<S>	+x)+x
q_1	<S>)+<S>	x)+x
q_1	<B>)+<S>	x)+x
q_1	x)+<S>	x)+x
q_1	)+<S>	)+x
q_1	+<S>	+x
q_1	<S>	x
q_1	<B>	x
q_1	x	x
q_1		
q		

Bottom-up parsing

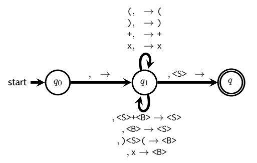
- Stack holds a word in $(N \cup \Sigma)^*$ such that it is possible to derive the part of the input string that has been consumed from its reverse.
- · t any time, may read a letter from input string and push it on top of the stack
- · t any time, may non-deterministically choose a rule $::= \gamma_1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

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



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

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



State	Stack	·nput
q_0		(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>	
q_1	<B>+<B>	
q_1	<S>+<B>	
q_1	<S>	
q		

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).
  - · lgorithms for parsing any context free grammar in cubic[1] time, based on dynamic programming (Earley, and Cocke-Younger-Kasami).

[1]· lso sub-cubic galactic algorithms

Parsing overview

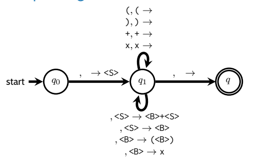
- Basic problem with both top-down and bottom-up construction: *non-determinism*
  - Non-deterministic search is inefficient
    - E.g., consider <S> ::= <S>a | <S>b | ϵ. Top-down parser must "guess" the entire input string at the beginning (breadth-first backtracking search takes exponential time in length of input string, depth-first does not terminate).
  - · lgorithms 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:** ·· (· eft-to-right, · eftmost derivation) parsers: top-down
    - Easy to understand & write by hand
  - **Next time:** *· R* (· eft-to-right, *R*ightmost derivation) parsers: bottom-up
    - More general, (variations) implemented in parser generators

[1]· lso sub-cubic galactic algorithms

LL parsing

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

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



- "· ny time top of the stack is a non-terminal , non-deterministically choose a production ::= $\gamma \in R$. Pop 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 when 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 $S \Rightarrow^* \quad \beta$, for any word $w \in \Sigma^k$, if there is some $\quad ::= \gamma \in R$ 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 when 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 $S \Rightarrow^* \quad \beta$, for any word $w \in \Sigma^k$, if there is some $\quad ::= \gamma \in R$ 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 DF· $\rightarrow$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

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

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

Eliminating left recursion

- · grammar is **left-recursive** if there is a non-terminal such that $\Rightarrow^+$ γ (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[illegible]) parsers

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

Mechanical construction of LL[illegible]) parsers

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

Mechanical construction of LL⊠) parsers

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

Computing nullable

- **nullable** is the *smallest set* of non-terminals such that if there is some $::= \gamma_1 ... \gamma_n \in R$ with $\gamma_1, ..., \gamma_n \in$ **nullable** implies $\in$ **nullable**
  - Fixpoint computation:
    - **nullable**$_0 = \emptyset$
    - **nullable**$_{i\ 1} = \{\ : \exists\ _1, ...,\ _n \in$ **nullable**$_i.\ ::=\ _1 ...\ _n \in R\}$
    - **nullable** $= \overset{\infty}{\underset{i=0}{}}$ **nullable**$_i$

```
nullable ← ∅;
changed ← true;
while changed do
    changed ← false;
    for  :=  1...  n ∈ R do
        if  ∉ nullable ∧  1, ...,  n ∈ nullable then
            nullable ← nullable ∪ {  };
            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 $::= \gamma_1 ... \gamma_i ... \gamma_n \in R$, with $\gamma_1, ..., \gamma_{i\ 1}$ nullable, **first**( ) $\supseteq$ **first**(γ_i)
- **follow** is the *smallest function* such that
  - For each $::= \gamma_1 ... \gamma_i ... \gamma_n \in R$, with $\gamma_{i\ 1}, ..., \gamma_n$ nullable, **follow**$(\gamma_i) \supseteq$ **follow**( )
  - For each $::= \gamma_1 ... \gamma_i ... \gamma_j ... \gamma_n \in R$, with $\gamma_{i\ 1}, ..., \gamma_{j\ 1}$ nullable, **follow**$(\gamma_i) \supseteq$ **first**( )
- Both can be computed using a fixpoint algorithm, like **nullable**

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