COS320: Compiling Techniques

Zak Kincaid

February 28, 2024

- HW3 on course webpage later today. Due March 25. **Start early!**
  - You will implement a compiler for a simple imperative programming language (Oat), targeting LLVMlite.
  - You may work individually or in pairs
- Midterm next Thursday
  - Covers material in lectures up to February 29th (this Thursday)
    - Interpreters, program transformation, X86, IRs, lexing, parsing
  - How to prepare:
    - Sample exams on Canvas later today
    - 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

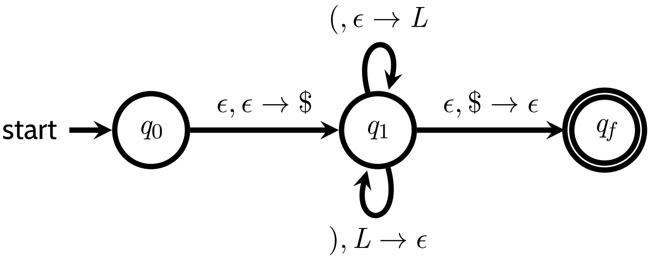
- A *context-free grammar* $G = (N, \Sigma, R, S)$ consists of:
  - N: a finite set of *non-terminal symbols*
  - Σ: a finite alphabet (or set of *terminal symbols*)
  - $R \subseteq N \times (N \cup \Sigma)^*$ a finite set of *rules* or *productions*
  - $S \in N$: the starting non-terminal.

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 word w is accepted by G if is derivable in zero or more steps from the starting non-terminal
    - Write $\gamma \Rightarrow \gamma'$ if γ' is obtained from γ by replacing a non-terminal symbol in γ with the right-hand-side of one of its rules
    - Write $\gamma \Rightarrow^* \gamma'$ if γ' can be obtained from γ using 0 or more derivation steps
    - A word $w \in \Sigma^*$ is accepted by G if $S \Rightarrow^* w$

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> $\mid$ (<S>) $\mid$ ϵ:
  - *Stack alphabet*: $\$$ marks bottom of the stack, L marks unbalanced left paren



Recall: pushdown automata

- A *push-down automaton* $A = (Q, \Sigma, \Gamma, \Delta, s, F)$ consists of
  - Q: a finite set of states
  - Σ: an (input) alphabet
  - Γ: a (stack) alphabet
  - $\Delta \subseteq \underbrace{Q}_{\text{source}} \times \underbrace{(\Sigma \cup \{\epsilon\})}_{\text{read input}} \times \underbrace{\Gamma^*}_{\text{read stack}} \times \underbrace{Q}_{\text{dest}} \times \underbrace{\Gamma^*}_{\text{write stack}}$, the transition relation
  - $s \in Q$: start state
  - $F \subseteq Q$: set of final (accepting) states

Recall: pushdown automata

- A *push-down automaton* $A = (Q, \Sigma, \Gamma, \Delta, s, F)$ consists of
  - Q: a finite set of states
  - Σ: an (input) alphabet
  - Γ: a (stack) alphabet
  - $\Delta \subseteq \underbrace{Q}_{\text{source}} \times \underbrace{(\Sigma \cup \{\epsilon\})}_{\text{read input}} \times \underbrace{\Gamma^*}_{\text{read stack}} \times \underbrace{Q}_{\text{dest}} \times \underbrace{\Gamma^*}_{\text{write stack}}$, the transition relation
  - $s \in Q$: start state
  - $F \subseteq Q$: set of final (accepting) states
- A word w is accepted by A if there is a w-labeled accepting path in A
  - A *configuration* of A is a pair (q, v) consisting of a state $q \in Q$ and a stack $v \in \Gamma^*$
  - Write $(q, v) \xrightarrow{w} (q', v')$ if there is some $t \in \Gamma^*$ such that $v = at$, $v' = bt$, and $(q, w, a, q', b) \in \Delta$
  - Write $(q, v) \xrightarrow{w}{}^* (q', v')$ if there is some $w_1, \dots, w_n$ and $(q_1, v_1), \dots, (q_{n-1}, v_{n-1})$ such that $w = w_1 \cdots w_n$ and

$$(q, v) \xrightarrow{w_1} (q_1, v_1) \xrightarrow{w_2} (q_2, v_2) \xrightarrow{w_3} \dots \xrightarrow{w_{n-1}} (q_{n-1}, v_{n-1}) \xrightarrow{w_n} (q', v')$$

  - A word w is accepted iff $(s, \epsilon) \xrightarrow{w}{}^* (q, v)$ for some $q \in F$, $v \in \Gamma^*$.

Context free languages

- Claim: a language is recognized by a context-free grammar if and only if it is recognized by a pushdown automaton
  - Say that a language is *context free* if it is recognized by a context-free grammar (equiv. pushdown automaton).
- Consequence: can "compile" context-free grammars to pushdown automata in order to implement parsers

Context free languages

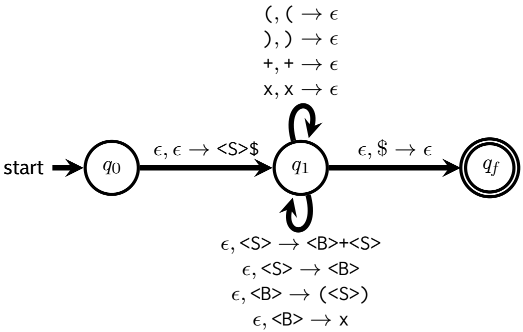
- Claim: a language is recognized by a context-free grammar if and only if it is recognized by a pushdown automaton
  - Say that a language is *context free* if it is recognized by a context-free grammar (equiv. pushdown automaton).
- Consequence: can "compile" context-free grammars to pushdown automata in order to implement parsers
- Two strategies, which correspond to different ways to implement parsers:
  - Top-down (LL parsing)
  - Bottom-up (LR parsing)

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

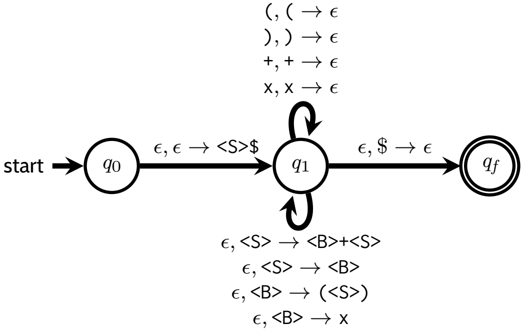
<S> ::= <B>+<S> | <B>

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



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

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



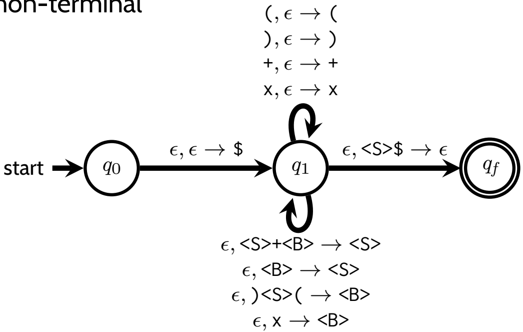
State	Stack	Input
q_0	ϵ	(x+x)+x
q_1	<S>$	(x+x)+x
q_1	<B>+<S>$	(x+x)+x
q_1	(<S>)+<S>$	(x+x)+x
q_1	<S>)+<S>$	x+x)+x
q_1	<B>+<S>)+<S>$	x+x)+x
q_1	x+<S>)+<S>$	x+x)+x
q_1	+<S>)+<S>$	+x)+x
q_1	<S>)+<S>$	x)+x
q_1	<B>)+<S>$	x)+x
q_1	x)+<S>$	x)+x
q_1	)+<S>$	)+x
q_1	+<S>$	+x
q_1	<S>$	x
q_1	<B>$	x
q_1	x$	x
q_1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

Bottom-up parsing

- Stack holds a word in $(N \cup \Sigma)^*$ such that it is possible to derive the part of the input string that has been consumed from its reverse.
- 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 $A ::= \gamma_1 \dots \gamma_n$ and apply it in reverse: pop $\gamma_n \dots \gamma_1$ off the top of the stack, and push A.
- Accept when stack just contains start non-terminal

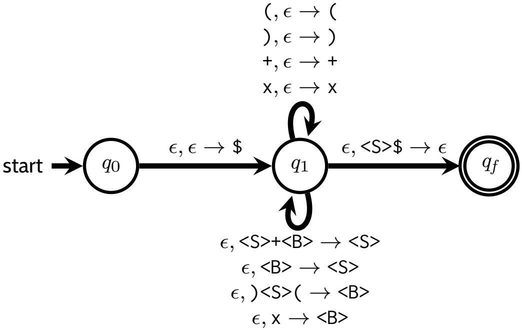
<S> ::= <B>+<S> | <B>

<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: Valiant 1975

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 time:** *LR* (*L*eft-to-right, *R*ightmost derivation) parsers: bottom-up
    - More general, (variations) implemented in parser generators

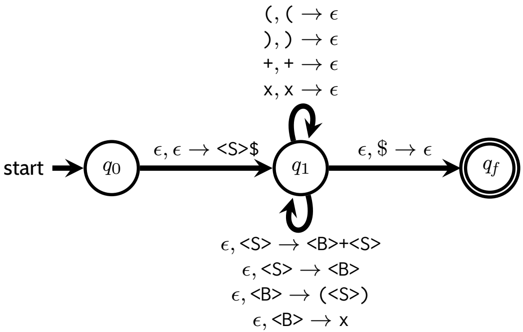
---

[1] Also sub-cubic galactic algorithms: Valiant 1975

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 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 $S \Rightarrow^* \alpha A \beta$, 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

- 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 $S \Rightarrow^* \alpha A \beta$, 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 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 $S \Rightarrow^* \alpha A \beta$, 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$\to$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

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**$(A) = \{a \in \Sigma : \exists \gamma, \gamma' . S \Rightarrow \gamma A a \gamma'\}$

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 δ for G can be computed using **first**, **follow**, and **nullable**:
  1. For each non-terminal A and letter a, initialize $\delta(A, a)$ to $\emptyset$
  2. For each rule $A ::= \gamma$
     - Add γ to $\delta(A, a)$ for each $a \in$ **first**(γ)
     - If γ is nullable, add γ to $\delta(A, a)$ 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 δ for G can be computed using **first**, **follow**, and **nullable**:
  1. For each non-terminal A and letter a, initialize $\delta(A, a)$ to $\emptyset$
  2. For each rule $A ::= \gamma$
     - Add γ to $\delta(A, a)$ for each $a \in$ **first**(γ)
     - If γ is nullable, add γ to $\delta(A, a)$ for each $a \in$ **follow**(A)
- G is $LL(1)$ iff $\delta(A, a)$ 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 δ for G can be computed using **first**, **follow**, and **nullable**:
  1. For each non-terminal A and letter a, initialize $\delta(A, a)$ to $\emptyset$
  2. For each rule $A ::= \gamma$
     - Add γ to $\delta(A, a)$ for each $a \in$ **first**(γ)
     - If γ is nullable, add γ to $\delta(A, a)$ for each $a \in$ **follow**(A)
- G is $LL(1)$ iff $\delta(A, a)$ 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) $\delta(A, a)$ (or reject if $\delta(A, a)$ 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 $\leftarrow \emptyset$;
changed $\leftarrow$ true;
while *changed* **do**
    changed $\leftarrow$ false;
    for $A := \gamma_1 \ldots \gamma_n \in R$ **do**
        if $A \notin$ *nullable* $\wedge\ \gamma_1, \ldots, \gamma_n \in$ *nullable* **then**
            nullable $\leftarrow$ *nullable* $\cup \{A\}$;
            changed $\leftarrow$ true;

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

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