CS 444/644 Course Notes

Compiler Construction

Ondřej Lhoták • Winter 2019 • University of Waterloo

Last Revision: January 17, 2019

Table of Contents

Abstract

These notes are intended as a resource for myself; past, present, or future students of this course, and anyone interested in the material. The goal is to provide an end-to-end resource that covers all material discussed in the course displayed in an organized manner. These notes are my interpretation and transcription of the content covered in lectures. The instructor has not verified or confirmed the accuracy of these notes, and any discrepancies, misunderstandings, typos, etc. as these notes relate to course's content is not the responsibility of the instructor. If you spot any errors or would like to contribute, please contact me directly.

1 January 7, 2019

1.1 Basic overview of a compiler

A **compiler** takes a source language and translates it to a target language. The source language could be, for example, C, Java, or JVM bytecode and the target language could be, for example, machine language or JVM bytecode.
A compiler could be divided into two parts:

Front-end analysis Front-end could be further divided into two parts: the first being scanning and parsing (assignment 1) and the second being context-sensitive analysis (assignment 2,3,4).

Some refer to context-sensitive analysis as "middle-end".

Back-end synthesis The backend could also be divided into two parts: the first being optimization (CS 744) and the second being code generation (assignment 5).

1.2 Overview of front-end analysis

Goal: is the input a valid program? An auxiliary step is to also generate information about the program for use in synthesis later on.
There are several steps in the front-end:

Scanning Split sequence of characters into sequence of tokens. Each token consists of its lexeme (actual characters) and its kind.

There are tools for generating the DFA expressions from a regular language e.g. lex.

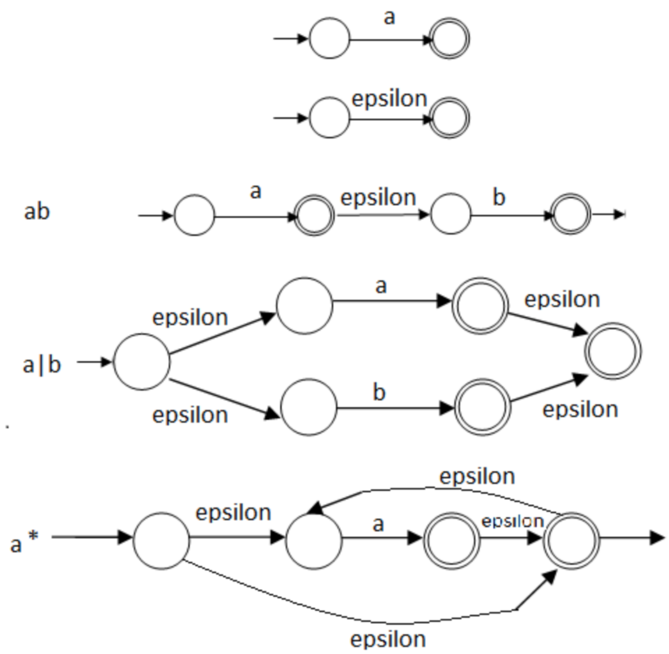
2 January 9, 2019

2.1 Scanning tools (lex)

Our goal is to specify our grammar in terms of regular expressions (regex) which lex can convert into a scanning DFA.
Review of regex to language (set of words):

RE	$L(e)$
$\varnothing$	$\{\}$
ϵ	$\{\epsilon\}$
$a \in \Sigma$	$\{a\}$
$e_1 e_2$	$\{xy \mid x \in L(e_1), y \in L(e_2)\}$
$e_1 \mid e_2$	$L(e_1) \cup L(e_2)$
e^*	$L(\epsilon \mid e \mid ee \mid eee \mid \ldots)$

The corresponding NFAs we can construct per each regex rule are



Let Σ be the set of characters, Q the set of states, q_0 the initial state, A the accepting states, and δ the transition function, an NFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$ where

$$\begin{aligned}&\text{NFA} : \delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q \text{ (subset of Q)}\\&\text{DFA} : \delta : Q \times \Sigma \to Q\end{aligned}$$

i.e. the transition function of an NFA returns a subset of states in Q.
We note in our above NFAs some accepting states are equivalent (connected by an ϵ transition).
We define

Definition 2.1 (ϵ-closure I)**.** The ϵ-closure(S) of a set of states S is the set of states reachable from S by (0 or more) ϵ-transitions.

Another equivalent recursive definition

Definition 2.2 (ϵ-closure II)**.** Smallest set S' such that

$$\begin{aligned}&S' \supseteq S\\&S' \supseteq \{q \mid q' \in S', q \in \delta(q', \epsilon)\}\end{aligned}$$

We note that any NFA can be converted in a corresponding DFA. The input is an NFA $(\Sigma, Q, q_0, A, \delta)$ and we'd like to get a DFA $(\Sigma, Q', q_0', A', \delta')$ where

$$\begin{aligned}&q_0' = \epsilon\text{-closure}(\{q_0\})\\&\delta'(q', a) = \epsilon\text{-closure}\Big(\bigcup_{q \in q'} \delta(q, a)\Big)\end{aligned}$$

we note that each state of our DFA is a set of states in the original NFA e.g. $\{1, 2, 4\}$ may be a state in the DFA.

We generate more states in our DFA by iterating through every $a \in \Sigma$ and applying rule two to our initial state q'_0. We do this until no further states can be generated from existing DFA states and all $a \in \Sigma$ have been exhausted. We note that the entire set of states Q' in the DFA can be recursively defined as the smallest set of subsets of Q such that

$$\begin{aligned} &Q' \supseteq \{q'_0\} \\ &Q' \supseteq \{\delta'(q', a) \mid q' \in Q'\} \qquad \forall a \in \Sigma \end{aligned}$$

We note that if any accepting state is included in a state of the DFA, we can accept it (since we can reach the corresponding accepting state in the NFA). Thus

$$A' = \{q' \in Q' \mid q' \cap A \neq \varnothing\}$$

3 January 14, 2019

3.1 DFA recognition scanning

The algorithm for using a DFA to recognize if a word is valid in the grammar

Algorithm 1 DFA recognition

input word w, DFA $M = (\Sigma, Q, q_0, \delta A)$
output boolean $w \in L(M)$?

```
q ← q_0
for i from 1 to |w| do
    q ← δ(q, w[i])
return q ∈ A
```

Scanning is similar where we take a *sequence of symbols* and convert it into a *sequence of tokens* using a DFA. In **maximal munch** we have

Algorithm 2 Maximal munch (abstract)

input DFA M specifying language L of valid tokens, string of symbols w
output sequence of tokens, each token $\in L$ that concantenates to w

```
while until end of output do
    Find a maximal prefix of remaining input that is in L
    ERROR if no non-empty prefix in L
```

note that **maximal munch** takes the **maximal prefix**: if we do not take the maximal prefix we may run into ambiguity. A more concrete implementation

Algorithm 3 Maximal munch (concrete)

```
while until end of output do
    Run DFA and record last seen accepting state until it gets stuck (or it transitions to ERROR state)
    Backtrack DFA and the input to last seen accepting state
    ERROR if there is no accepting state
    Output prefix as the next token
    Set DFA back to start state
```

Note that in Java which uses maximal munch, $a - -b$ would be parsed as $a(- - b)$ which would return a parsing error (as opposed to $a - (-b)$ since $--$ is a token in Java).

3.2 Constructing the scanning DFA

The overall algorithm to convert regular expressions denoting tokens to a scanning DFA is

Algorithm 4 Regex to scanning DFA

input REs $R_1, \ldots, R_n$ for token kinds in priority order
output DFA with accepting states labelled with token kinds

1: Construct an NFA M for $R_1 \mid R_2 \mid \ldots \mid R_n$
2: Convert NFA to DFA M' (each state of M' is set of states in M)
3: For each accepting state of M', output highest priority token kind of the set of NFA accepting states

4 January 16, 2019

4.1 Context-free grammar

Regular expressions are great for scanning but would not work for an arbitrary depth of tokens when it comes to parsing grammar.
To address this we define **context-free grammars** which uses recursion to specify arbitrary structures in the grammar at an arbitrary depth in the parse tree.

Definition 4.1 (Context free grammar)**.** A **context-free grammar** is a 4-tuple $G = (N, T, R, S)$ where we have (NB: notation for each class)

Terminals T (e.g. a, b, c; denoted with lowercase alphabet)

Non-terminals N (e.g. A, B, C, S; denoted with uppercase alphabet)

Symbols The set of symbols are $V = N \cup T$ (e.g. W, X, Y, Z)

String of terminals T^* (e.g. w, x, y, z)

String of symbols V^* (e.g. α, β, γ)

Production rules $R \subseteq N \times V^*$ (e.g. $A \to \alpha$)

Start non-terminal S

Definition 4.2 (Directly derives)**.** $\beta A \gamma \Rightarrow \beta \alpha \gamma$ if $A \to \alpha \in R$: that is $\beta A \gamma$ **directly derives** $\beta \alpha \gamma$.

Definition 4.3 (Derives)**.** $\alpha \Rightarrow^* \beta$ if $\alpha \Rightarrow \gamma$ and $\gamma \Rightarrow^* \beta$: that is α **derives** β.

Definition 4.4 (Sentential form)**.** α is a **sentential form** if $S \Rightarrow^* \alpha$.

Definition 4.5 (Sentence)**.** x is a **sentence** if $x \in T^*$ and x is a sentential form.

Definition 4.6 (Language)**.** $L(G) = \{x \in T^* \mid S \Rightarrow^* x\}$ is the **language generated by** G (set of sentences).

4.2 Recognizer and parsing

The task of a **recognizer** is to see if $x \in L(G)$ for some grammar G.
The task of **parsing** is to find a derivation from S to x.

Example 4.1. Suppose we have a grammar G with $R = \{A \to BgC, B \to ab, C = ef\}, S = A, N = \{A, B, C\}, T = \{a, b, e, f, g\}$.
We can derive the following sentence

$$A \Rightarrow BgC \Rightarrow abgC \Rightarrow abgef$$

notice this is the only sentence in $L(G)$.
We could represent this as a tree where A is at the root, B, g, C are each children of A (3 children), a, b and e, f are children of B and C, respectively.
Note that for this particular grammar, we may have multiple derivations for $abgef$ (we could have expanded C first) but we have one unique parse tree.

Definition 4.7 (Ambiguous grammar)**.** A grammar is **ambiguous** if $\exists > 1$ parse tree for the same sentence.

Definition 4.8 (Left(Right) derivation)**.** In a **left(right) derivation** we always expand the left(right)-most non-terminal.

There is a *one-to-one correspondence* between parse trees, left derivations, and right derivations: that is given a sentence with a unique parse tree, it has a unique left derivation and a unique right derivation.

4.3 Top-down parser

The simplest approach for parsing a sentence x from S (start symbol) is using a **top-down parser**:

Algorithm 5 Top-down parser

```
1: α ← S
2: while α ≠ x do
3:     Replace first non-terminal A in α with β, assuming A → β ∈ R
```

Replacing the *first* non-terminal results in a left derivation.

4.4 LL(1) parser

Example 4.2. Given a grammar with the following production rules

$$E \to aE'$$
$$E' \to +a$$
$$E' \to \epsilon$$

Suppose we wanted to derive $a + a$. Intuitively we have $E \Rightarrow aE' \Rightarrow a + a$. Our intuition told us to use $E' \to +a$ instead of $E' \to' \epsilon$.

To improve on our top-down parsing, we introduce the LL(1) parser:

Algorithm 6 LL(1) parser

$\alpha \leftarrow S$
while $\alpha \neq x$ **do**
    Let A be the first non-terminal in α ($\alpha = yA\gamma$)
    Let a be the first terminal after y in x ($x = ya\zeta$)
    $A \rightarrow \beta \leftarrow \text{predict}(A, a)$
    Replace A with β in α

where $\text{predict}(A, a)$ follows our intuition of picking the rule with A on the LHS that works best when a is the next terminal: our **lookahead**.