

richardwu.ca

CS 444/644 COURSE NOTES

COMPILER CONSTRUCTION

ONDŘEJ LHOTÁK • WINTER 2019 • UNIVERSITY OF WATERLOO

 Last Revision: January 24, 2019

Table of Contents

1	January 7, 2019	1
1.1	Basic overview of a compiler	1
1.2	Overview of front-end analysis	1
2	January 9, 2019	1
2.1	Scanning tools (lex)	1
3	January 14, 2019	3
3.1	DFA recognition scanning	3
3.2	Constructing the scanning DFA	4
4	January 16, 2019	4
4.1	Context-free grammar	4
4.2	Recognizer and parsing	5
4.3	Top-down parser	5
5	January 21, 2019	5
5.1	LL(1) parser	5
5.2	First, nullable and follow sets	7
6	January 23, 2019	7
6.1	Note on LL(k) parsers	7
6.2	Bottom-up parsing	7
6.3	LR(0) parser	8

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

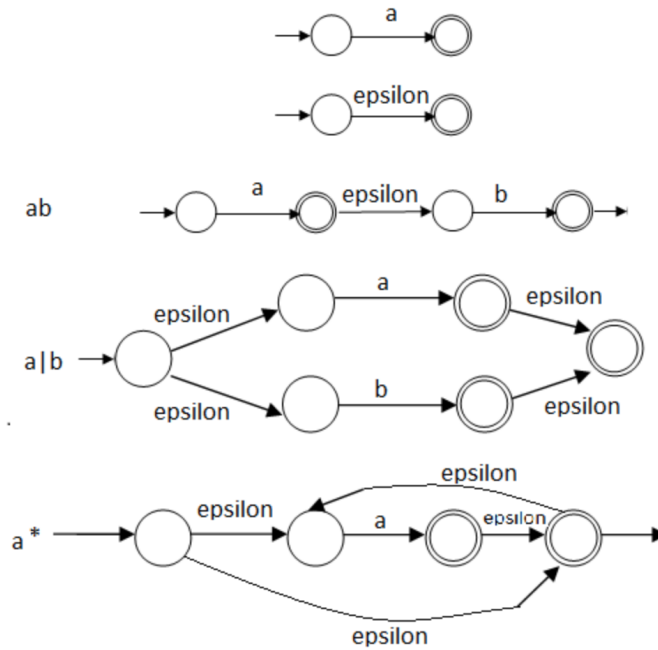
2.1 Scanning tools (lex)

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	$L(e)$
\emptyset	$\{\}$
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^*	$L(\epsilon \mid e \mid ee \mid eee \mid \dots)$

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

$$\text{NFA} : \delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q \text{ (subset of } Q)$$

$$\text{DFA} : \delta : Q \times \Sigma \rightarrow Q$$

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

$$S' \supseteq S$$

$$S' \supseteq \{q \mid q' \in S', q \in \delta(q', \epsilon)\}$$

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

$$q'_0 = \epsilon\text{-closure}(\{q_0\})$$

$$\delta'(q', a) = \epsilon\text{-closure}\left(\bigcup_{q \in q'} \delta(q, a)\right)$$

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

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

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

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

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 $w \in L(M)$?

```

1:  $q \leftarrow q_0$ 
2: for  $i$  from 1 to  $|w|$  do
3:    $q \leftarrow \delta(q, w[i])$ 
4: return  $q \in A$ 
```

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 concatenates to w

```

1: while until end of output do
2:   Find a maximal prefix of remaining input that is in  $L$ 
3:   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)

```

1: while until end of output do
2:   Run DFA and record last seen accepting state until it gets stuck (or it transitions to ERROR state)
3:   Backtrack DFA and the input to last seen accepting state
4:   ERROR if there is no accepting state
5:   Output prefix as the next token
6:   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, \dots, R_n for token kinds in priority order

output DFA with accepting states labelled with token kinds

- 1: Construct an NFA M for $R_1 \mid R_2 \mid \dots \mid R_n$
 - 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. $A \rightarrow \alpha$)

Start non-terminal S

Definition 4.2 (Directly derives). $\beta A \gamma \Rightarrow \beta \alpha \gamma$ if $A \rightarrow \alpha \in R$: 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 $R = \{A \rightarrow BgC, B \rightarrow ab, C \rightarrow ef\}$, $S = A$, $N = \{A, B, C\}$, $T = \{a, b, e, f, g\}$.

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: $\alpha \leftarrow S$
 - 2: **while** $\alpha \neq x$ **do**
 - 3: Replace **first** non-terminal A in α with β , assuming $A \rightarrow \beta \in R$
-

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

5 January 21, 2019

5.1 LL(1) parser

Example 5.1. Given a grammar with the following production rules

$$\begin{aligned} E &\rightarrow aE' \\ E' &\rightarrow +a \\ E' &\rightarrow \epsilon \end{aligned}$$

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

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

Algorithm 6 LL(1) parser

input x is the input string to parse

- 1: $\alpha \leftarrow S$
 - 2: **while** $\alpha \neq x$ **do**
 - 3: Let A be the first non-terminal in α ($\alpha = yA\gamma$)
 - 4: Let a be the first terminal after y in x ($x = ya\zeta$)
 - 5: $A \rightarrow \beta \leftarrow \text{predict}(A, a)$
 - 6: 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**.

Remark 5.1. The first “L” represents scanning the **input from left to right**; the second “L” denotes **left-canonical derivation** (always expanding the leftmost non-terminal); the “1” denotes a 1 **symbol lookahead**

Definition 5.1 (Augmented grammar). In order to make our 1 lookahead algorithm work with the start rule, we need to augment it:

1. Add a fresh terminal “\$”
2. Add production $S' \rightarrow S\$$ where S' is our new start and S was the start of our original grammar.

We also need to append “\$” to our input. Our augmented grammar becomes

$$\begin{aligned} S' &\rightarrow E\$ \\ E &\rightarrow aE' \\ E' &\rightarrow +a \\ E' &\rightarrow \epsilon \end{aligned}$$

If we implement $LL(1)$ naively with string replacements we could see that the algorithm runs in $O(n^2)$. To do this in $O(n)$ time we could utilize a stack for our α derivation. Our revised algorithm proceeds as

Algorithm 7 LL(1) parser

input x is the input string to parse

- 1: Push $S\$$ onto stack
 - 2: **for** a in $x\$$ **do**
 - 3: **while** top of stack is a non-terminal A **do**
 - 4: Pop A
 - 5: Find $A \rightarrow \beta$ in $\text{predict}(A, a)$ or ERROR
 - 6: Push β
 - 7: Pop b , terminal on top of stack
 - 8: **if** $b \neq a$ **then**
 - 9: ERROR
-

We define

$$\text{predict}(A, a) = \{A \rightarrow \beta \in R \mid \exists \gamma \quad \beta \Rightarrow^* a\gamma \text{ or } (\beta \Rightarrow^* \epsilon \text{ and } \exists \gamma, \delta \quad S' \Rightarrow^* \gamma A a \delta)\}$$

A grammar is $LL(1)$ iff $|\text{predict}(A, a)| \leq 1$ for all A, a .

5.2 First, nullable and follow sets

Definition 5.2 (First set). If $\beta \Rightarrow^* a\gamma$ then $a \in \text{first}(\beta)$ where $\text{first}(\beta)$ is the **first set** of β .

Definition 5.3 (Nullable set). If $\beta \Rightarrow^* \epsilon$ then β is **nullable**.

Definition 5.4 (Follow set). If $S' \Rightarrow^* \gamma A a \delta$ then $a \in \text{follow}(A)$ where $\text{follow}(A)$ is the **follow set** of A .

Note that in our above example grammar, we have

$$\text{nullable} = \{\epsilon\}$$

and

$$\begin{aligned}\text{first}(\epsilon) &= \{\} \\ \text{first}(+a) &= \{+\} \\ \text{first}(aE') &= \{a\} \\ \text{first}(E\$') &= \{a\}\end{aligned}$$

In general to compute the **follow set**, we define it as

1. If $B \rightarrow \alpha A \gamma$ then $\text{first}(\gamma) \subseteq \text{follow}(A)$
2. If $B \rightarrow \alpha A \gamma$ and $\text{nullable}(\gamma)$ then $\text{follow}(B) \subseteq \text{follow}(A)$

6 January 23, 2019

6.1 Note on $\text{LL}(k)$ parsers

Note that in $\text{LL}(k)$ parsers we perform leftmost derivation. This means that the sentence $3 - 2 - 1$ would be parsed (with parentheses denoting derivations/subtrees) $3 - (2 - 1)$ which would incorrectly result in 2.

Ideally we wanted $(3 - 2) - 1$ or **left associative**. In general, $\text{LL}(k)$ parser cannot parse arbitrary left associative languages.

Remark 6.1. One could imagine an $\text{LL}(1)$ parser as generating the parse tree from top to bottom, left to right where the parse tree is rooted at S and the leaves are the tokens in the input x .

An alternative to $\text{LL}(k)$ parsers which is a top-down parser (from start symbol S to input) are bottom-up parsers.

6.2 Bottom-up parsing

Example 6.1. Suppose we have the same grammar as before

$$\begin{aligned}S &\rightarrow E\$ \\ E &\rightarrow E + a \\ E &\rightarrow a\end{aligned}$$

Suppose we wanted to parse the sentence $a + a\$$. We proceed in a bottom up fashion

$$\begin{aligned}a + a\$ &\Leftarrow E + a\$ \\ &\Leftarrow E\$ \\ &\Leftarrow S\end{aligned}$$

where we scan from left to right the tokens in the input and reduce with “intuition”.

Remark 6.2. Given a parse tree rooted at S with leaves as tokens of input x , a bottom-up parser generates the parse tree from the bottom-left corner and moves upwards and rightwards until the start symbol is reached.

proceed to look at a particular bottom-up parser.

6.3 LR(0) parser

The best way to implement these parsers is to think about invariants and ensuring our algorithm always maintains the invariants:

LL(k) invariant The invariant for an LL(k) parser is that

$$\begin{aligned} S &\Rightarrow^* \alpha \\ S &\Rightarrow^* \text{seen input} + \text{stack} \end{aligned}$$

i.e. S always derives the string of processed symbols we’ve derived so far.

LR(k) invariant Here the invariant is that

$$\begin{aligned} \alpha &\Rightarrow^* x \\ \text{stack} + \text{unseen input} &\Rightarrow^* x \end{aligned}$$

that is our current string of processed symbols plus the rest of the unseen input should be able to derive our entire input. Equivalently

$$\begin{aligned} S &\Rightarrow^* \alpha \\ S &\Rightarrow^* \text{stack} + \text{unseen input} \end{aligned}$$

in other words we require the stack to always be a viable prefix:

Definition 6.1 (Viable prefix). α is a **viable prefix** if it is a prefix of some sentential form i.e.

$$\exists \beta \text{ s.t. } S \Rightarrow^* \alpha\beta$$

In LR(0) parsing as above, we keep a stack of symbols we have processed so far. We have the following algorithm

Algorithm 8 LR(0) parser

```

1: for  $a$  in  $x\$$  do
2:   while  $\text{Reduce}(\text{stack}) = \{A \rightarrow \gamma\}$  do
3:     Pop  $\gamma$  off stack (i.e. pop  $|\gamma|$  tokens)
4:     Push  $A$  onto stack
5:   if  $\text{Reject}(\text{stack} + a)$  then ERROR
6:   Push  $a$  onto stack

```

where we define

$$\text{Reduce}(\alpha) = \{A \rightarrow \gamma \mid \exists \beta \text{ s.t. } \alpha = \beta\gamma \text{ and } \beta A \text{ is a VP}\}$$

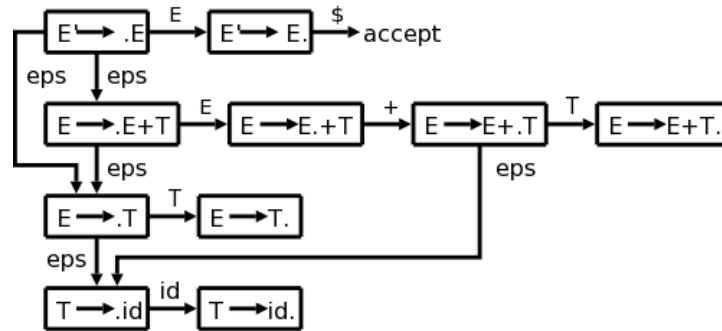
where VP stands for **viable prefix** (see above) and $\text{Reject}(\alpha)$ is true if α is not a VP.

Question 6.1. How do we check if α is a VP? Note that the set of *all* VPs is itself a context-free language.

Exercise 6.1. Given a CFG G , construct a CFG G' for the set of VPs of G .

Remark 6.3. Only some VPs α 's occur during the algorithm, thus we can build a simpler NFA for VPs that actually occur.

We can construct the NFA (such as the following) systematically from a given grammar



We define the construction of the LR(0) NFA as follows:

$$\Sigma = T \cup N$$

$$Q = \{A \rightarrow \alpha \cdot \beta \mid A \rightarrow \alpha\beta \in R\}$$

$$q_0 = S' \rightarrow S\$$$

$$A = Q$$

all states are accepting i.e. denote input is a VP

$$\delta(A \rightarrow \alpha \cdot B\beta, \epsilon) = \{B \rightarrow \cdot \gamma \mid B \rightarrow \gamma \in R\}$$

$$\delta(A \rightarrow \alpha \cdot X\beta, X) = \{A \rightarrow \alpha X \cdot \beta\}$$

If the NFA ends up in state $A \rightarrow \gamma \cdot$ on input $\beta\gamma$ then $\text{Reduce}(\beta\gamma)$ contains $A \rightarrow \gamma$ because the NFA would also accept βA .