# <span id="page-0-0"></span>Compiler concepts: Parsing COMSM1302 Overview of Computer Architecture

John Lapinskas, University of Bristol

# Describing languages

We have a description of Hack syntax in Nisan and Schocken, right?

#### Describing languages

#### We have a description of Hack syntax in Nisan and Schocken, right?



#### machine that not an  $\log$  of the Hack platform. These symbols can be completely ignored for now, we specify them for completeness.

women, need that programs can need don from a light-oast and display **SPERIES:** ARR (1882) Programs can that displays a basis of a display display display display that an a series. The series and the layer<br>basis display to the series and the layer of the series with the couple<br>of the serie

Label optimize (Lieds on appear anywhere in a Hock assumbly program and are declared using the system area (The decrease) and the animalism of the model of the property behavior in the program. Contained and the system of

deds. Any qualid our appearing in a Rink annuality program definitions operator as a<br>spaceting in a Black mustafy program of the local state of the state<br>of the state of the state of the state of the state of the local state<br>of the state of the state of the state of the state of

#### Charges Humiling

induces platform can be connected to two periphend UC

whether, a Constantine, a label declaration, or a

on it a test line of the form typolool. The assuming<br>intuities by building conduction that the same of the test<br>opposs. This is the only action that the monoider rates<br>of declaration; no binary code to generated. That's wh  $\label{eq:1} \infty \mbox{ non-simons reduced to } m \mbox{, non-simons, for } \alpha \in \mathbb{N}$  <br> which only at the symbolic break generating to sock.

Senator and materials are computed to the first project of the project states of the senator of the se

Community. A next line beginning with two shahes (e) and ending at the end of the line is established a extension and is ignored.

White space Looking space characters and empty lines are ignored

#### Maybe this isn't the most convenient form possible...

John Lapinskas [Parsing](#page-0-0) 2 / 12

is can openly momey boots<br>on tableaust using  $\alpha\gamma$  states. The symbols property<br>of the first function scheme addresses and could see any distinction of<br>the property problem and could see any distinction  $\epsilon$  Then are several hireb of probables symbols,  $\times$  consistent and modelity of low-level Hirds

The intermedian gas with the a regularizer  $N_{\rm c}$  and gas<br>transfer three regularizations (  $N_{\rm c}$  and  $N_{\rm c}$  <br>is the fitting of the same of matrix  $N_{\rm c}$  is the same of the<br>polarization (  $N_{\rm c}$  is the same of t really made working variables, sometimes artistrative as virtual eggineers

get, or gaves. The resideds priceting them will be and the search in part of the result in part in our field in

Constant and sembols: These are the cor's in Alestracisms of the fort

**Case assessings:** All the asset<br>by manusias  $\frac{1}{2}$  and  $\frac{1}{2}$  within in upper<br>sec. By convertion,  $\frac{1}{2}$  adding any vectors in upper<br>sec, and randele cyclots is becomes for  $\frac{2}{2}$  and<br> $\frac{2}{2}$  for examples.

#### 43 Bob Programming

We now turn to proceed three cooraples of low-decuty<br>programming, using the Hoof moresticy largesque, Since project 4 focus<br>on an acting Hock morestic programs, it will now you would be a<br>moreledy real and understand thes

Example 1. Figure 4.6 above a program that adds up the values of the fits run. Kidsk registers, adds 17 to the same, and mores the sensition for this fits that this state and the state of the same per a test sample) is eq

We'll need a highly sophisticated mathematical construction:

#### A more sophisticated approach: Madlibs!

We'll need a highly sophisticated mathematical construction: Madlibs.



Source: Jesse Vig via Medium [\(here\)](https://towardsdatascience.com/a-i-plays-mad-libs-and-the-results-are-terrifying-78fa44e7f04e)

## A more sophisticated approach: Madlibs?

We'll need a highly sophisticated mathematical construction: Madlibs.



A context-free grammar (or just grammar) is a way of quickly and rigorously specifying which strings in a language have valid syntax.

There is a deep and rich mathematical theory here, which we thankfully don't need to learn! Programmers express grammars in **Backus-Naur** Form (BNF), and usually just understanding BNF is enough.

BNF is basically Madlibs, but recursive.

## Introduction to Madlibs Backus-Naur Form (BNF)

Here's a simple example:

```
⟨noun⟩ ::= 'lecturer' | 'student' | 'pizza'
⟨presentVerb⟩ ::= 'eats' | 'devours' | 'consumes'
```
You can read each | as "or" and each  $\cdot :=$  as "is defined as". E.g. a  $\langle$ noun $\rangle$ is defined as one of the three strings 'lecturer', 'student', or 'pizza'.

## Introduction to Madlibs Backus-Naur Form (BNF)

Here's a simple example:

```
\langlenoun\rangle :: = 'lecturer' | 'student' | 'pizza'
⟨presentVerb⟩ ::= 'eats' | 'devours' | 'consumes'
```
You can read each | as "or" and each  $\cdot :=$  as "is defined as". E.g. a  $\langle$ noun $\rangle$ is defined as one of the three strings 'lecturer', 'student', or 'pizza'.

We can also build up definitions in terms of other definitions, e.g.

```
⟨sentence⟩ ::= 'The' ⟨noun⟩ ⟨presentVerb⟩ 'the' ⟨noun⟩
```
Here, valid ⟨sentence⟩s include:

```
⟨noun⟩ ::= 'lecturer' | 'student' | 'pizza'
⟨presentVerb⟩ ::= 'eats' | 'devours' | 'consumes'
```
You can read each | as "or" and each  $\cdot :=$  as "is defined as". E.g. a  $\langle$ noun $\rangle$ is defined as one of the three strings 'lecturer', 'student', or 'pizza'.

We can also build up definitions in terms of other definitions, e.g.

```
⟨sentence⟩ ::= 'The' ⟨noun⟩ ⟨presentVerb⟩ 'the' ⟨noun⟩
```
Here, valid ⟨sentence⟩s include:

```
'The' ⟨noun⟩ ⟨presentVerb⟩ 'the' ⟨noun⟩
```

```
⟨noun⟩ ::= 'lecturer' | 'student' | 'pizza'
⟨presentVerb⟩ ::= 'eats' | 'devours' | 'consumes'
```
You can read each | as "or" and each  $\cdot :=$  as "is defined as". E.g. a  $\langle$ noun $\rangle$ is defined as one of the three strings 'lecturer', 'student', or 'pizza'.

We can also build up definitions in terms of other definitions, e.g.

```
⟨sentence⟩ ::= 'The' ⟨noun⟩ ⟨presentVerb⟩ 'the' ⟨noun⟩
```
Here, valid ⟨sentence⟩s include:

'The' 'lecturer' 'consumes' 'the' 'pizza'

```
⟨noun⟩ ::= 'lecturer' | 'student' | 'pizza'
⟨presentVerb⟩ ::= 'eats' | 'devours' | 'consumes'
```
You can read each | as "or" and each  $\cdot :=$  as "is defined as". E.g. a  $\langle$ noun $\rangle$ is defined as one of the three strings 'lecturer', 'student', or 'pizza'.

We can also build up definitions in terms of other definitions, e.g.

```
⟨sentence⟩ ::= 'The' ⟨noun⟩ ⟨presentVerb⟩ 'the' ⟨noun⟩
```
Here, valid ⟨sentence⟩s include:

```
'The' 'student' 'eats' 'the' 'pizza'
```

```
\langlenoun\rangle :: = 'lecturer' | 'student' | 'pizza'
⟨presentVerb⟩ ::= 'eats' | 'devours' | 'consumes'
```
You can read each | as "or" and each  $\cdot :=$  as "is defined as". E.g. a  $\langle$ noun $\rangle$ is defined as one of the three strings 'lecturer', 'student', or 'pizza'.

We can also build up definitions in terms of other definitions, e.g.

```
⟨sentence⟩ ::= 'The' ⟨noun⟩ ⟨presentVerb⟩ 'the' ⟨noun⟩
```
Here, valid ⟨sentence⟩s include:

'The' 'lecturer' 'devours' 'the' 'student'

```
⟨noun⟩ ::= 'lecturer' | 'student' | 'pizza'
⟨presentVerb⟩ ::= 'eats' | 'devours' | 'consumes'
```
You can read each | as "or" and each  $\cdot :=$  as "is defined as". E.g. a  $\langle$ noun $\rangle$ is defined as one of the three strings 'lecturer', 'student', or 'pizza'.

We can also build up definitions in terms of other definitions, e.g.

```
⟨sentence⟩ ::= 'The' ⟨noun⟩ ⟨presentVerb⟩ 'the' ⟨noun⟩
```
Here, valid ⟨sentence⟩s include:

#### 'The' 'lecturer' 'devours' 'the' 'student'

Anything we define as part of the grammar must be enclosed in  $\Diamond$ s. We call these **non-terminal symbols**. Anything else (e.g. 'lecturer') is a terminal symbol or token.

The last feature of BNF is the source of its power: it allows **recursion**.

For example, suppose our tokens are '0' through '9', and we want to define a non-terminal symbol that matches precisely the non-negative whole numbers (allowing for leading zeroes). We could write:

The last feature of BNF is the source of its power: it allows **recursion**.

For example, suppose our tokens are '0' through '9', and we want to define a non-terminal symbol that matches precisely the non-negative whole numbers (allowing for leading zeroes). We could write:

$$
\langle \text{digit} \rangle ::= \text{'}0' | \text{'}1' | \text{'}2' | \text{'}3' | \text{'}4' | \text{'}5' | \text{'}6' | \text{'}7' | \text{'}8' | \text{'}9' \langle \text{number} \rangle ::= \langle \text{digit} \rangle | \langle \text{digit} \rangle \langle \text{number} \rangle
$$

E.g. '016' is a ⟨number⟩ because we can expand ⟨number⟩'s definition as:

$$
\langle number \rangle \longrightarrow \langle digit \rangle \langle number \rangle \longrightarrow \langle digit \rangle \langle digit \rangle \langle number \rangle
$$
  

$$
\longrightarrow \langle digit \rangle \langle digit \rangle \langle digit \rangle \longrightarrow '0' '1' '6'.
$$

The last feature of BNF is the source of its power: it allows **recursion**.

For example, suppose our tokens are '0' through '9', and we want to define a non-terminal symbol that matches precisely the non-negative whole numbers (allowing for leading zeroes). We could write:

$$
\langle \text{digit} \rangle ::= \text{'}0' | \text{'}1' | \text{'}2' | \text{'}3' | \text{'}4' | \text{'}5' | \text{'}6' | \text{'}7' | \text{'}8' | \text{'}9' \langle \text{number} \rangle ::= \langle \text{digit} \rangle | \langle \text{digit} \rangle \langle \text{number} \rangle
$$

E.g. '016' is a ⟨number⟩ because we can expand ⟨number⟩'s definition as:

$$
\langle number \rangle \longrightarrow \langle digit \rangle \langle number \rangle \longrightarrow \langle digit \rangle \langle digit \rangle \langle number \rangle
$$
  

$$
\longrightarrow \langle digit \rangle \langle digit \rangle \langle digit \rangle \longrightarrow '0' '1' '6'.
$$

That's it! That's all of BNF. It can be hard to use and hard to reason about, but the syntax is simple.

#### Example: Better integers

How should we redefine  $\langle$  number $\rangle$  to allow negative numbers, but forbid leading zeroes? (Assume we have '0' through '9' and '-' as tokens.)

## Example: Better integers

How should we redefine  $\langle$  number $\rangle$  to allow negative numbers, but forbid leading zeroes? (Assume we have '0' through '9' and '–' as tokens.)

Some sanity checks for any such re-definition:

- $\bullet$  ' $-$ ' '1' '0' should be a  $\langle$ number $\rangle$ .
- '0' should be a ⟨number⟩.
- '0' '1' shouldn't be a ⟨number⟩.
- '−' '0' shouldn't be a ⟨number⟩.

### Example: Better integers

How should we redefine  $\langle$  number $\rangle$  to allow negative numbers, but forbid leading zeroes? (Assume we have '0' through '9' and '–' as tokens.)

Some sanity checks for any such re-definition:

- $\bullet$  ' $-$ ' '1' '0' should be a  $\langle$ number $\rangle$ .
- '0' should be a ⟨number⟩.
- '0' '1' shouldn't be a ⟨number⟩.
- '−' '0' shouldn't be a ⟨number⟩.

There are multiple approaches — there's no such thing as the "right" expression of a grammar in BNF. Here's one way:

```
\langleposDigit\rangle ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
⟨posNumber⟩ ::= ⟨posDigit⟩ | ⟨posNumber⟩ ⟨posDigit⟩ | ⟨posNumber⟩ '0'
    ⟨number⟩ ::= ⟨posNumber⟩ | '0' | '−' ⟨posNumber⟩
```
#### Parse trees

 $\langle posDigit \rangle ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'$ ⟨posNumber⟩ ::= ⟨posDigit⟩ | ⟨posNumber⟩ ⟨posDigit⟩ | ⟨posNumber⟩ '0' ⟨number⟩ ::= ⟨posNumber⟩ | '0' | '−' ⟨posNumber⟩

The goal of parsing is to convert a list of tokens into a **parse tree** or **concrete** syntax tree (CST) which gives its BNF structure. E.g. for "−886":



Each non-terminal symbol is a node. Its children are its BNF expansion, in order from left to right — so the leaves are precisely the tokens.

#### Abstract syntax trees

 $\langle \text{posDigit} \rangle ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'$ ⟨posNumber⟩ ::= ⟨posDigit⟩ | ⟨posNumber⟩ ⟨posDigit⟩ | ⟨posNumber⟩ '0' ⟨number⟩ ::= ⟨posNumber⟩ | '0' | '−' ⟨posNumber⟩

For efficiency and convenience, we may choose to process a CST into an **abstract** syntax tree (AST), which contains the same information in a more convenient form. E.g. we might decide we don't need the  $\langle$  posDigit $\rangle$  nodes:



Consider this grammar for simple arithmetic expressions.

⟨expression⟩ ::= ⟨number⟩ | ⟨expression⟩ ⟨operator⟩ ⟨expression⟩ | '(' ⟨expression⟩ ⟨operator⟩ ⟨expression⟩ ')' ⟨operator⟩ ::= '+' | '−' | '∗' | '/' | 'ˆ'

Consider this grammar for simple arithmetic expressions.

⟨expression⟩ ::= ⟨number⟩ | ⟨expression⟩ ⟨operator⟩ ⟨expression⟩ | '(' ⟨expression⟩ ⟨operator⟩ ⟨expression⟩ ')' ⟨operator⟩ ::= '+' | '−' | '∗' | '/' | 'ˆ'

Then a parser could output several valid CSTs for e.g.  $(3 + 4) * (5 - 1)/3$ . This **ambiguity** can be dealt with as long as the semantic meaning is not ambiguous. E.g. here it is the same for all CSTs.

## Generating parse trees

Parsing is a difficult and subtle problem, but a well-understood one.

## Generating parse trees

Parsing is a difficult and subtle problem, but a well-understood one.



NOOOOOOOOOOOOO!!!! YOU CAN'T HAVE THE COMPUTER WRITE YOUR PARSER FOR YOU!!! WHAT ABOUT NON-CONTEXT-FREE GRAMMARS AND RECURSIVE DESCENT AND ALL THE BEAUTIFUL SUBTLETY OF TYPE THEONO NOOOOOOOOOOOOO

haha yacc go brrrrrrrr

Source: Generated with imgflip [\(here\)](https://imgflip.com/memegenerator/234051219/nooo-haha-go-brrr).

This means we shouldn't try to solve it again ourselves! We should instead use a **parser generator** which takes our grammar in BNF form and outputs code for a parser in a language of our choice. (E.g. yacc for C.)

# <span id="page-25-0"></span>Extended Backus-Naur Form (EBNF)

Often both parser generators and language specifications add extra syntax to BNF for usability, but there's no one standard. Based loosely on ISO 14977, we'll add:

- ()s mean grouped terms, e.g.  $('0' | '1') ('0' | '1')$  means 00, 01, 10 or 11.
- $\lceil \cdot \rceil$ s mean optional terms, e.g.  $\lceil \cdot 0 \rceil \rceil \rceil$  means 01 or 1.
- $\{\}$ s mean repetition, e.g.  $\{0' | '1' \}$  means any number of zeroes and ones (including the empty string).
- $\bullet$  A B means anything that matches A, but doesn't match B, e.g.  $\langle$ number $\rangle$  –  $\langle$ posNumber $\rangle$  means any number that's not a  $\langle$ posNumber $\rangle$ .

This doesn't let BNF express any grammars it couldn't before (why not?), but it does make it much nicer to read and write. For example:

$$
\langle \text{digit} \rangle ::= \text{'}0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
$$
  

$$
\langle \text{number} \rangle ::= (['-'] (\langle \text{digit} \rangle - '0') \{ \langle \text{digit} \rangle \}) | '0'
$$

With EBNF, we can build a readable grammar for all of C, never mind Hack! See for example [here](http://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/cminusminusspec.html) (credit Samuya Debray).