Compiler concepts: Parsing
COMSM1302 Overview of Computer Architecture

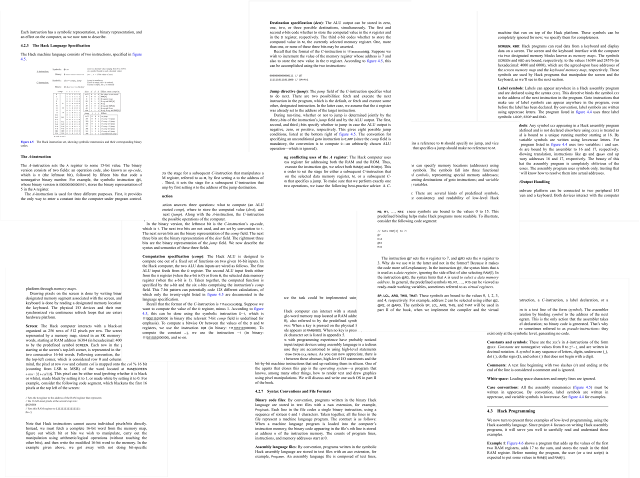
John Lapinskas, University of Bristol

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?



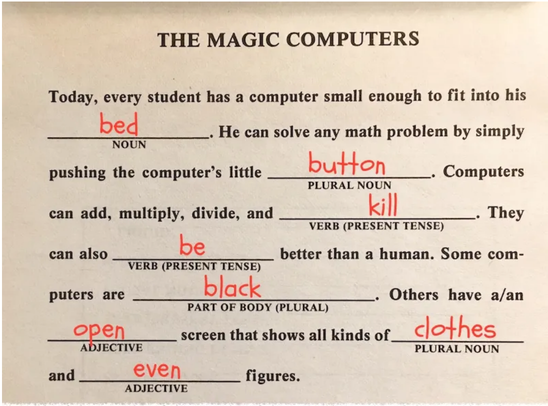
Maybe this isn't the most convenient form possible...

A more sophisticated approach

We'll need a highly sophisticated mathematical construction:

A more sophisticated approach: Madlibs!

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



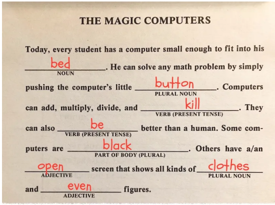
THE MAGIC COMPUTERS

Today, every student has a computer small enough to fit into his
_____bed_____. He can solve any math problem by simply
NOUN
pushing the computer's little _____button_____. Computers
PLURAL NOUN
can add, multiply, divide, and _____kill_____. They
VERB (PRESENT TENSE)
can also _____be_____ better than a human. Some com-
VERB (PRESENT TENSE)
puters are _____black_____. Others have a/an
PART OF BODY (PLURAL)
_____open_____ screen that shows all kinds of _____clothes_____
ADJECTIVE PLURAL NOUN
and _____even_____ figures.
ADJECTIVE

Source: Jesse Vig via Medium (here)

A more sophisticated approach: Madlibs?

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



THE MAGIC COMPUTERS

Today, every student has a computer small enough to fit into his
bed
NOUN
. He can solve any math problem by simply
pushing the computer's little
button
PLURAL NOUN
. Computers
can add, multiply, divide, and
kill
VERB (PRESENT TENSE)
. They
can also
be
VERB (PRESENT TENSE)
better than a human. Some com-
puters are
black
PART OF BODY (PLURAL)
. Others have a/an
open
ADJECTIVE
screen that shows all kinds of
clothes
PLURAL NOUN
and
even
ADJECTIVE
figures.

Source: Jesse Vig via Medium (here)

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:

$$\langle\text{noun}\rangle ::= \text{‘lecturer’} \mid \text{‘student’} \mid \text{‘pizza’}$$
$$\langle\text{presentVerb}\rangle ::= \text{‘eats’} \mid \text{‘devours’} \mid \text{‘consumes’}$$

You can read each | as "or" and each ::= as "is defined as". E.g. a ⟨noun⟩ is defined as one of the three strings ‘lecturer’, ‘student’, or ‘pizza’.

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

Here's a simple example:

$$\langle\text{noun}\rangle ::= \text{`lecturer'} \mid \text{`student'} \mid \text{`pizza'}$$
$$\langle\text{presentVerb}\rangle ::= \text{`eats'} \mid \text{`devours'} \mid \text{`consumes'}$$

You can read each $\mid$ as "or" and each $::=$ as "is defined as". E.g. a $\langle\text{noun}\rangle$ 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.

$$\langle\text{sentence}\rangle ::= \text{`The'}\ \langle\text{noun}\rangle\ \langle\text{presentVerb}\rangle\ \text{`the'}\ \langle\text{noun}\rangle$$

Here, valid $\langle\text{sentence}\rangle$s include:

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

Here's a simple example:

$$\langle\text{noun}\rangle ::= \text{`lecturer'} \mid \text{`student'} \mid \text{`pizza'}$$
$$\langle\text{presentVerb}\rangle ::= \text{`eats'} \mid \text{`devours'} \mid \text{`consumes'}$$

You can read each | as "or" and each ::= as "is defined as". E.g. a ⟨noun⟩ 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.

$$\langle\text{sentence}\rangle ::= \text{`The'}\ \langle\text{noun}\rangle\ \langle\text{presentVerb}\rangle\ \text{`the'}\ \langle\text{noun}\rangle$$

Here, valid ⟨sentence⟩s include:

$$\text{`The'}\ \langle\text{noun}\rangle\ \langle\text{presentVerb}\rangle\ \text{`the'}\ \langle\text{noun}\rangle$$

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

Here's a simple example:

$$\langle\text{noun}\rangle ::= \text{`lecturer'} \mid \text{`student'} \mid \text{`pizza'}$$
$$\langle\text{presentVerb}\rangle ::= \text{`eats'} \mid \text{`devours'} \mid \text{`consumes'}$$

You can read each | as "or" and each ::= as "is defined as". E.g. a ⟨noun⟩ 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.

$$\langle\text{sentence}\rangle ::= \text{`The'}\ \langle\text{noun}\rangle\ \langle\text{presentVerb}\rangle\ \text{`the'}\ \langle\text{noun}\rangle$$

Here, valid ⟨sentence⟩s include:

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

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

Here's a simple example:

$$\langle\text{noun}\rangle ::= \text{`lecturer'} \mid \text{`student'} \mid \text{`pizza'}$$
$$\langle\text{presentVerb}\rangle ::= \text{`eats'} \mid \text{`devours'} \mid \text{`consumes'}$$

You can read each | as "or" and each ::= as "is defined as". E.g. a ⟨noun⟩ 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.

$$\langle\text{sentence}\rangle ::= \text{`The'}\ \langle\text{noun}\rangle\ \langle\text{presentVerb}\rangle\ \text{`the'}\ \langle\text{noun}\rangle$$

Here, valid ⟨sentence⟩s include:

'The' **'student' 'eats'** 'the' **'pizza'**

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

Here's a simple example:

$$\langle\text{noun}\rangle ::= \text{`lecturer'} \mid \text{`student'} \mid \text{`pizza'}$$
$$\langle\text{presentVerb}\rangle ::= \text{`eats'} \mid \text{`devours'} \mid \text{`consumes'}$$

You can read each $|$ as "or" and each $::=$ as "is defined as". E.g. a $\langle\text{noun}\rangle$ 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.

$$\langle\text{sentence}\rangle ::= \text{`The'}\ \langle\text{noun}\rangle\ \langle\text{presentVerb}\rangle\ \text{`the'}\ \langle\text{noun}\rangle$$

Here, valid $\langle\text{sentence}\rangle$s include:

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

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

Here's a simple example:

$$\langle\text{noun}\rangle ::= \text{`lecturer'} \mid \text{`student'} \mid \text{`pizza'}$$
$$\langle\text{presentVerb}\rangle ::= \text{`eats'} \mid \text{`devours'} \mid \text{`consumes'}$$

You can read each | as "or" and each ::= as "is defined as". E.g. a ⟨noun⟩ 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.

$$\langle\text{sentence}\rangle ::= \text{`The'}\ \langle\text{noun}\rangle\ \langle\text{presentVerb}\rangle\ \text{`the'}\ \langle\text{noun}\rangle$$

Here, valid ⟨sentence⟩s include:

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

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

Example: Integers

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:

Example: Integers

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:

$$\begin{aligned}\langle\text{digit}\rangle &::= \text{‘0’} \mid \text{‘1’} \mid \text{‘2’} \mid \text{‘3’} \mid \text{‘4’} \mid \text{‘5’} \mid \text{‘6’} \mid \text{‘7’} \mid \text{‘8’} \mid \text{‘9’}\\ \langle\text{number}\rangle &::= \langle\text{digit}\rangle \mid \langle\text{digit}\rangle\ \langle\text{number}\rangle\end{aligned}$$

E.g. ‘016’ is a $\langle$number$\rangle$ because we can expand $\langle$number$\rangle$’s definition as:

$$\begin{aligned}\langle\text{number}\rangle &\longrightarrow \langle\text{digit}\rangle\ \langle\text{number}\rangle \longrightarrow \langle\text{digit}\rangle\ \langle\text{digit}\rangle\ \langle\text{number}\rangle\\ &\longrightarrow \langle\text{digit}\rangle\ \langle\text{digit}\rangle\ \langle\text{digit}\rangle \longrightarrow \text{‘0’ ‘1’ ‘6’}.\end{aligned}$$

Example: Integers

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:

$$\begin{aligned} \langle \text{digit} \rangle &::= \text{‘0’} \mid \text{‘1’} \mid \text{‘2’} \mid \text{‘3’} \mid \text{‘4’} \mid \text{‘5’} \mid \text{‘6’} \mid \text{‘7’} \mid \text{‘8’} \mid \text{‘9’} \\ \langle \text{number} \rangle &::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \ \langle \text{number} \rangle \end{aligned}$$

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

$$\begin{aligned} \langle \text{number} \rangle &\longrightarrow \langle \text{digit} \rangle \ \langle \text{number} \rangle \longrightarrow \langle \text{digit} \rangle \ \langle \text{digit} \rangle \ \langle \text{number} \rangle \\ &\longrightarrow \langle \text{digit} \rangle \ \langle \text{digit} \rangle \ \langle \text{digit} \rangle \longrightarrow \text{‘0’ ‘1’ ‘6’}. \end{aligned}$$

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 ⟨number⟩ to allow negative numbers, but forbid leading zeroes? (Assume we have ‘0’ through ‘9’ and ‘$-$’ as tokens.)

Example: Better integers

How should we redefine ⟨number⟩ 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:

- '$-$' '1' '0' should be a ⟨number⟩.
- '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 $\langle\text{number}\rangle$ 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:

- '$-$' '1' '0' should be a $\langle\text{number}\rangle$.
- '0' should be a $\langle\text{number}\rangle$.
- '0' '1' shouldn't be a $\langle\text{number}\rangle$.
- '$-$' '0' shouldn't be a $\langle\text{number}\rangle$.

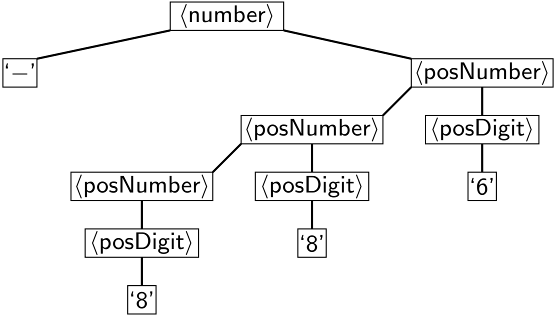
There are multiple approaches — there's no such thing as the "right" expression of a grammar in BNF. Here's one way:

$$\begin{aligned}\langle\text{posDigit}\rangle &::= \text{`1'} \mid \text{`2'} \mid \text{`3'} \mid \text{`4'} \mid \text{`5'} \mid \text{`6'} \mid \text{`7'} \mid \text{`8'} \mid \text{`9'}\\ \langle\text{posNumber}\rangle &::= \langle\text{posDigit}\rangle \mid \langle\text{posNumber}\rangle\ \langle\text{posDigit}\rangle \mid \langle\text{posNumber}\rangle\ \text{`0'}\\ \langle\text{number}\rangle &::= \langle\text{posNumber}\rangle \mid \text{`0'} \mid \text{`}-\text{'}\ \langle\text{posNumber}\rangle\end{aligned}$$

Parse trees

$$\begin{aligned}\langle\text{posDigit}\rangle &::= \text{`1'} \mid \text{`2'} \mid \text{`3'} \mid \text{`4'} \mid \text{`5'} \mid \text{`6'} \mid \text{`7'} \mid \text{`8'} \mid \text{`9'}\\ \langle\text{posNumber}\rangle &::= \langle\text{posDigit}\rangle \mid \langle\text{posNumber}\rangle\ \langle\text{posDigit}\rangle \mid \langle\text{posNumber}\rangle\ \text{`0'}\\ \langle\text{number}\rangle &::= \langle\text{posNumber}\rangle \mid \text{`0'} \mid \text{`}-\text{'}\ \langle\text{posNumber}\rangle\end{aligned}$$

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":

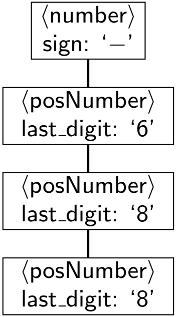


Each non-terminal symbol is a node. Its children are its BNF expansion, in order from left to right — so the leaves are precisely the tokens.

Abstract syntax trees

$$\begin{aligned}
\langle\text{posDigit}\rangle &::= \text{`1'} \mid \text{`2'} \mid \text{`3'} \mid \text{`4'} \mid \text{`5'} \mid \text{`6'} \mid \text{`7'} \mid \text{`8'} \mid \text{`9'} \\
\langle\text{posNumber}\rangle &::= \langle\text{posDigit}\rangle \mid \langle\text{posNumber}\rangle\ \langle\text{posDigit}\rangle \mid \langle\text{posNumber}\rangle\ \text{`0'} \\
\langle\text{number}\rangle &::= \langle\text{posNumber}\rangle \mid \text{`0'} \mid \text{`}-\text{'}\ \langle\text{posNumber}\rangle
\end{aligned}$$

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 ⟨posDigit⟩ nodes:



Ambiguity

Consider this grammar for simple arithmetic expressions.

$$\begin{aligned}\langle\text{expression}\rangle &::= \langle\text{number}\rangle \mid \langle\text{expression}\rangle\ \langle\text{operator}\rangle\ \langle\text{expression}\rangle \mid \\ &\qquad \text{`('}\ \langle\text{expression}\rangle\ \langle\text{operator}\rangle\ \langle\text{expression}\rangle\ \text{`)'} \\ \langle\text{operator}\rangle &::= \text{`}+\text{'} \mid \text{`}-\text{'} \mid \text{`}*\text{'} \mid \text{`}/\text{'} \mid \text{`\^{}'}\end{aligned}$$

Ambiguity

Consider this grammar for simple arithmetic expressions.

$$\begin{aligned}\langle\text{expression}\rangle &::= \langle\text{number}\rangle \mid \langle\text{expression}\rangle\ \langle\text{operator}\rangle\ \langle\text{expression}\rangle \mid \\ &\qquad\qquad\qquad\quad \text{‘(’}\ \langle\text{expression}\rangle\ \langle\text{operator}\rangle\ \langle\text{expression}\rangle\ \text{‘)’} \\ \langle\text{operator}\rangle &::= \text{‘}+\text{’} \mid \text{‘}-\text{’} \mid \text{‘}*\text{’} \mid \text{‘}/\text{’} \mid \text{‘\^{}’}\end{aligned}$$

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).

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.)

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.
- []s mean optional terms, e.g. [‘0’] ‘1’ means 01 or 1.
- {}s mean repetition, e.g. {‘0’ | ‘1’} means any number of zeroes and ones (including the empty string).
- $A - B$ means anything that matches A, but doesn't match B, e.g. ⟨number⟩ − ⟨digit⟩ means any number that's not a ⟨digit⟩.[1]

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:

$$\begin{aligned}\langle\text{digit}\rangle &::= \text{‘0’} \mid \text{‘1’} \mid \text{‘2’} \mid \text{‘3’} \mid \text{‘4’} \mid \text{‘5’} \mid \text{‘6’} \mid \text{‘7’} \mid \text{‘8’} \mid \text{‘9’}\\ \langle\text{number}\rangle &::= ([\text{‘}-\text{’}]\ (\langle\text{digit}\rangle - \text{‘0’})\ \{\langle\text{digit}\rangle\}) \mid \text{‘0’}\end{aligned}$$

With EBNF, we can build a readable grammar for all of C, never mind Hack! See for example here (credit Samuya Debray).

[1] This is only valid EBNF when B expands into one of finitely many possible sequences of tokens, so e.g. ⟨number⟩ − ⟨posNumber⟩ wouldn't be valid.