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 us top of the Hack plotters. These symbols can be completely ignored for new; we specify them for completeness.

BOURD, BD: Dick program can note this from a kedword and display data an a score. The scores and its heyboard introduct with the compare-tion are adopted memory Model income an accuracy map. The public location and other as board, aspectively, is the visual to NHM and 200° its hexadedimit. A000 and 00000, which are the apped-spot here addresses of the scores measure may and the locational memory may program. The supporting were strained by the strain and program of the scores republic are used by links, programs the manipulate the scores and the laphond, see visual to vision.

Label random, Label can appear anywhere in a Hack assembly program and are derived using the corone zoro. This directive back det evolve into us the address of the next interaction in the program. Each interactions the mode and of the low random starting in the program. Each interaction the mode are of fabel symbols can again anywhere in the program, are instituted to their hows how constrained to program. Each interaction with a gaptome lowers. The program listed in figure 4.1 uses three hold symbols colors, this main allocal.

serverion, a Construction, a label declaration, or a

White some Loaders sense characters and every loss are invested

Case convertients: All the anothly memories (figure 4.7) must be written in apperate, its converties, bod worked are written in processes and until the controls in how reasons. Say how I () for examples

We now turn to proper three complex of low-low-ly-segmenting, using the likele susceibly inequage. Since project 4 focuses on writing likele assembly program, it will some you will to carefully used and understand these

Example 1. Figure 4.5 shows a program that adds up the values of the first run EAM registers, adds 17 to the sum, and source the sensit in the third EAM register. Before running the program, the user (or a test sample) to copyright-part sums values in MARQ and MRR().

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

John Lapinskas

Parsing



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)

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 h |
| He can solve any math problem by simpl |
| pushing the computer's little Compute |
| can add, multiply, divide, and The |
| can also be better than a human. Some con |
| puters are PART OF RODY OFLURALD . Others have a/a |
| ADJECTIVE screen that shows all kinds of |
| and even 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 noun \rangle ::= 'lecturer' | 'student' | 'pizza' 
\langle presentVerb \rangle ::= 'eats' | 'devours' | 'consumes'
```

You can read each | as "or" and each ::= 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:

```
\langle noun \rangle ::= 'lecturer' | 'student' | 'pizza' 
\langle presentVerb \rangle ::= 'eats' | 'devours' | 'consumes'
```

You can read each | as "or" and each ::= 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.

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

Here, valid $\langle sentence \rangle s$ include:

```
\langle noun \rangle ::= 'lecturer' | 'student' | 'pizza' 
\langle presentVerb \rangle ::= 'eats' | 'devours' | 'consumes'
```

You can read each | as "or" and each ::= 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.

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

Here, valid $\langle sentence \rangle s$ include:

```
'The' \langle noun \rangle \; \langle presentVerb \rangle 'the' \langle noun \rangle
```

```
\langle noun \rangle ::= 'lecturer' | 'student' | 'pizza' 
\langle presentVerb \rangle ::= 'eats' | 'devours' | 'consumes'
```

You can read each | as "or" and each ::= 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.

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

Here, valid $\langle sentence \rangle s$ include:

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

```
\langle noun \rangle ::= 'lecturer' | 'student' | 'pizza' 
\langle presentVerb \rangle ::= 'eats' | 'devours' | 'consumes'
```

You can read each | as "or" and each ::= 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.

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

Here, valid $\langle sentence \rangle s$ include:

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

```
\langle noun \rangle ::= 'lecturer' | 'student' | 'pizza' 
\langle presentVerb \rangle ::= 'eats' | 'devours' | 'consumes'
```

You can read each | as "or" and each ::= 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.

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

Here, valid $\langle sentence \rangle s$ include:

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

```
\langle noun \rangle ::= 'lecturer' | 'student' | 'pizza' 
\langle presentVerb \rangle ::= 'eats' | 'devours' | 'consumes'
```

You can read each | as "or" and each ::= 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.

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

Here, valid $\langle sentence \rangle s$ include:

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

Anything we define as part of the grammar must be enclosed in $\langle \rangle$ 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:

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

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

$$\begin{array}{l} \langle \mathsf{number} \rangle \longrightarrow \langle \mathsf{digit} \rangle \ \langle \mathsf{number} \rangle \longrightarrow \langle \mathsf{digit} \rangle \ \langle \mathsf{digit} \rangle \ \langle \mathsf{number} \rangle \\ \longrightarrow \langle \mathsf{digit} \rangle \ \langle \mathsf{digit} \rangle \ \langle \mathsf{digit} \rangle \longrightarrow `0` \ `1` \ `6`. \end{array}$$

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:

$$\begin{array}{l} \langle \text{digit} \rangle ::= `0' | `1' | `2' | `3' | `4' | `5' | `6' | `7' | `8' | `9' \\ \neg \text{number} \rangle ::= \langle \text{digit} \rangle | \langle \text{digit} \rangle \langle \text{number} \rangle \end{array}$$

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

$$\begin{array}{l} \langle \mathsf{number} \rangle \longrightarrow \langle \mathsf{digit} \rangle \ \langle \mathsf{number} \rangle \longrightarrow \langle \mathsf{digit} \rangle \ \langle \mathsf{digit} \rangle \ \langle \mathsf{number} \rangle \\ \longrightarrow \langle \mathsf{digit} \rangle \ \langle \mathsf{digit} \rangle \ \langle \mathsf{digit} \rangle \longrightarrow `0` \ `1` \ `6`. \end{array}$$

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:

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

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:

- '-' '1' '0' should be a $\langle number \rangle$.
- '0' should be a $\langle number \rangle$.
- '0' '1' shouldn't be a (number).
- '-' '0' shouldn't be a $\langle number \rangle$.

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

```
\begin{array}{l} \langle posDigit \rangle ::= `1' \mid `2' \mid `3' \mid `4' \mid `5' \mid `6' \mid `7' \mid `8' \mid `9' \\ \langle posNumber \rangle ::= \langle posDigit \rangle \mid \langle posNumber \rangle \; \langle posDigit \rangle \mid \langle posNumber \rangle \; `0' \\ \langle number \rangle ::= \langle posNumber \rangle \mid `0' \mid `-' \; \langle posNumber \rangle \end{array}
```

Parse trees

 $\begin{array}{l} \left< posDigit \right> ::= `1' | `2' | `3' | `4' | `5' | `6' | `7' | `8' | `9' \\ \left< posNumber \right> ::= \left< posDigit \right> | \left< posNumber \right> \left< posDigit \right> | \left< posNumber \right> `0' \\ \left< number \right> ::= \left< posNumber \right> | `0' | `-' \left< posNumber \right> \end{array}$

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.

| lohn | Lanineka |
|--------|------------|
| 301111 | Euphilokas |

Abstract syntax trees

 $\begin{array}{l} \langle \mathsf{posDigit} \rangle ::= `1' \mid `2' \mid `3' \mid `4' \mid `5' \mid `6' \mid `7' \mid `8' \mid `9' \\ \langle \mathsf{posNumber} \rangle ::= \langle \mathsf{posDigit} \rangle \mid \langle \mathsf{posNumber} \rangle \; \langle \mathsf{posDigit} \rangle \mid \langle \mathsf{posNumber} \rangle \; `0' \\ \langle \mathsf{number} \rangle ::= \langle \mathsf{posNumber} \rangle \mid `0' \mid `-' \; \langle \mathsf{posNumber} \rangle \\ \end{array}$

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.

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

Consider this grammar for simple arithmetic expressions.

$$\begin{split} \langle expression \rangle &::= \langle number \rangle \mid \langle expression \rangle \; \langle operator \rangle \; \langle expression \rangle \mid \\ & `(' \; \langle expression \rangle \; \langle operator \rangle \; \langle expression \rangle \; `)' \\ \langle operator \rangle & ::= `+' \mid `-' \mid `*' \mid '/' \mid ``` \end{split}$$

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.



N000000000000000000 CANT 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 N0000000000000

haha yacc go brrrrrrr

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) - (posNumber) means any number that's not a (posNumber).

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:

$$\begin{array}{l} \langle \text{digit} \rangle ::= `0' \mid `1' \mid `2' \mid `3' \mid `4' \mid `5' \mid `6' \mid `7' \mid `8' \mid `9' \\ \langle \text{number} \rangle ::= ([`-'] (\langle \text{digit} \rangle - `0') \{\langle \text{digit} \rangle \}) \mid `0' \end{array}$$

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