University of Waterloo

Parsers

Jon Eyolfson

June 3, 2013

Goals

Formulate a grammar for integer arithmetic

Create an AST and interpreter for our grammar

Our Grammar

Here is our attempt at the grammar without multiplication and division for now:

$$
\begin{aligned}
T &= \{+, -, \texttt{INT}\} \\
N &= \{e\} \\
S &= e \\
P &= \begin{cases} e \to e + e \\ e \to e - e \\ e \to \texttt{INT} \end{cases}
\end{aligned}
$$

Start AST

Abstract Syntax Trees

An AST is a tree representation of the input language

This tree should only represents the important details of the input

Unimportant Details in Our Grammar

Our parse trees have the following unimportant details:

- Rule names
- Brackets (or grouping)

Our AST Nodes

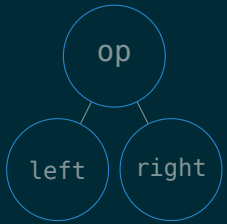
BinaryExpr need to know what operator to use, the left and right side
IntegerExpr to store the value of an integer

Both of which are **Expr** (which is an abstract base class)

Our AST Node Representations

We will represent our AST nodes as follows:

BinaryExpr



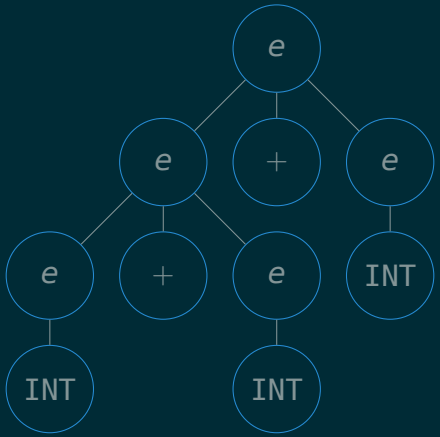
op is either $+, -, *$ or $/$
`left` and `right` are **Exprs**

IntegerExpr

`value` is an `int` (in Java we can pass the regular expression matched by the `INT` token to `Integer.parseInt`)

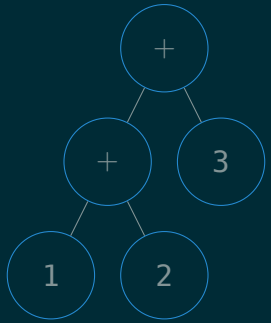
AST Question (1)

If our input is "1 + 2 + 3" and our parse tree is the following:



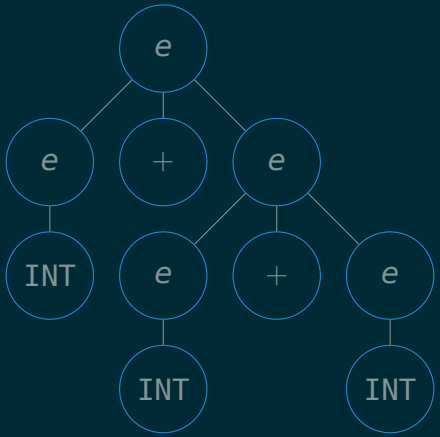
What is our AST?

AST Answer (1)



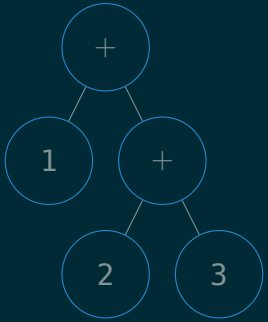
AST Question (2)

Now consider the same input and the following parse tree:



What is our AST now?

AST Answer (2)



End AST

Start Interpreter

Our Own Interpreter

An interpreter executes the AST directly

Our result should be equivalent to following all the mathematical rules for the input

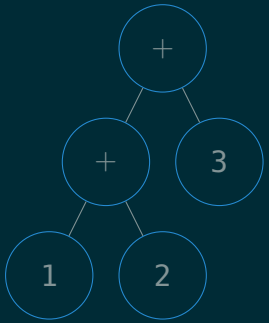
The rules in this case is the DMAS part of BEDMAS

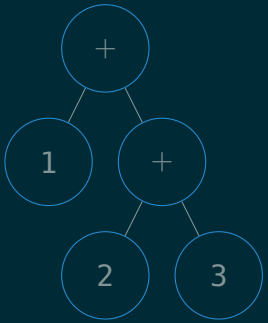
Our Interpreter Implementation

```
public int eval(BinaryExpr e) {
  switch (e.op) {
    case '*':
      return e.left.eval() * e.right.eval();
    case '/':
      return e.left.eval() / e.right.eval();
    case '+':
      return e.left.eval() + e.right.eval();
    case '-':
      return e.left.eval() - e.right.eval();
  }
}
public int eval(IntegerExpr i) {
  return i.value;
}
```

AST Evaluation (1)

We call `eval` with the root node, what is the result?





The result in both cases is 6

Ambiguity

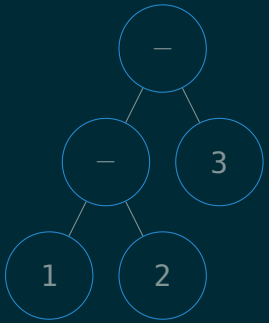
With ambiguity either parse tree/AST is correct

Therefore, the implementer can decide which one they want to use

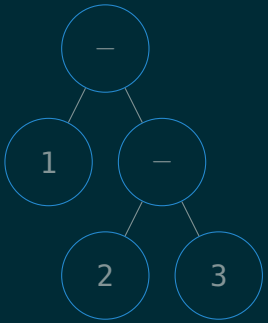
What happens if we use $-$ instead of $+$?

AST Evaluation (2)

Again, calling `eval` with the root node, what is the result?



The result is -4



The result is 2

Now, depending on the implementation you get different results

End Interpreter

Start Unambiguity

Removing Ambiguity

We have to enforce:

- Precedence
- Associativity

Precedence Rules

1. Each level of precedence should be it's own rule
2. The lowest level of precedence should be the top level rule (highest level of precedence is the last rule)
3. Each rule should use the next highest level of precedence
4. The highest level of precedence includes the operands

Our Grammar (Attempt 3)

Following the rules, our grammar is now:

$$
\begin{aligned}
T &= \{+, -, \texttt{INT}\} \\
N &= \{e, p\} \\
S &= e \\
P &= \begin{cases} e \to e + e \\ e \to e - e \\ e \to p \\ p \to \texttt{INT} \end{cases}
\end{aligned}
$$

But the grammar is still ambiguous

Associativity Rules

Consider e as our current level of precedence and p as the next highest

$e \to e$ 0P p, would make it left associative

$e \to p$ 0P e, would make it right associative

Our Grammar (Attempt 4)

We expect left associativity, our grammar is now:

$$
\begin{aligned}
T &= \{+, -, \texttt{INT}\} \\
N &= \{e, p\} \\
S &= e \\
P &= \begin{cases} e \to e + p \\ e \to e - p \\ e \to p \\ p \to \texttt{INT} \end{cases}
\end{aligned}
$$

Parsing

Let's use our previous grammar attempt

Parse the following input: `"1 - 2 - 3"`

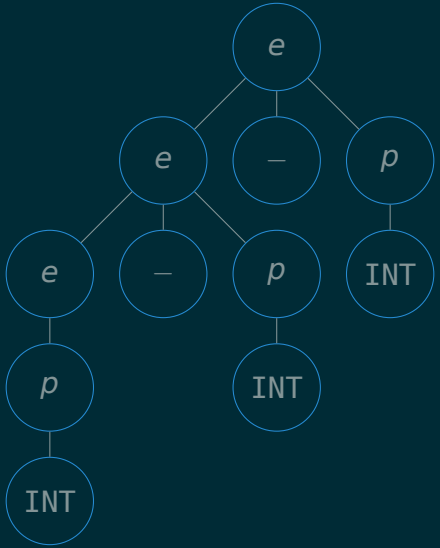
We have the following terminals: `INT` – `INT` – `INT`

Previous Derivation

$$
\begin{aligned}
e &\Rightarrow e - p \\
&\Rightarrow e - p - p \\
&\Rightarrow p - p - p \\
&\Rightarrow \texttt{INT} - p - p \\
&\Rightarrow \texttt{INT} - \texttt{INT} - p \\
&\Rightarrow \texttt{INT} - \texttt{INT} - \texttt{INT}
\end{aligned}
$$

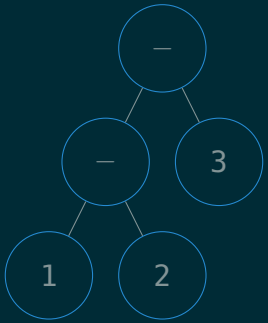
This is the only derivation

Previous Parse Tree



What about the corresponding AST?

Previous AST



Our interpreter would return `-4` as expected

Our Grammar (Attempt 5)

Let's add multiplication and division:

$$
\begin{aligned}
T &= \{+, -, *, /, \texttt{INT}\} \\
N &= \{e, p\} \\
S &= e \\
P &= \begin{cases}
e \to e + p \\
e \to e - p \\
e \to e * p \\
e \to e / p \\
e \to p \\
p \to \texttt{INT}
\end{cases}
\end{aligned}
$$

Parsing

Let's use our previous grammar attempt

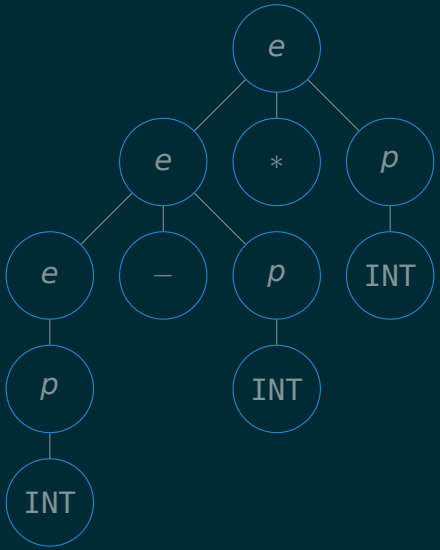
Parse the following input: `"1 - 2 * 3"`

We have the following terminals: `INT` $-$ `INT` $*$ `INT`

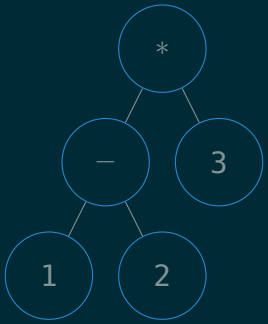
Previous Derivation

$$
\begin{aligned}
e &\Rightarrow e * p \\
&\Rightarrow e - p * p \\
&\Rightarrow p - p * p \\
&\Rightarrow \texttt{INT} - p * p \\
&\Rightarrow \texttt{INT} - \texttt{INT} * p \\
&\Rightarrow \texttt{INT} - \texttt{INT} * \texttt{INT}
\end{aligned}
$$

Previous Parse Tree



Previous AST



Our interpreter would return `-3` which is wrong

How would we fix this?

Precedence Rules

1. Each level of precedence should be it's own rule
2. The lowest level of precedence should be the top level rule (highest level of precedence is the last rule)
3. Each rule should use the next highest level of precedence
4. The highest level of precedence includes the operands

Our Grammar (Attempt 6)

We have to add another level of precedence (t) for multiplication and division:

$$
\begin{aligned}
T &= \{+, -, *, /, \texttt{INT}\} \\
N &= \{e, t, p\} \\
S &= e \\
P &= \begin{cases}
e \to e + t \\
e \to e - t \\
e \to t \\
t \to t * p \\
t \to t / p \\
t \to p \\
p \to \texttt{INT}
\end{cases}
\end{aligned}
$$

Parsing

Let's use our previous grammar attempt

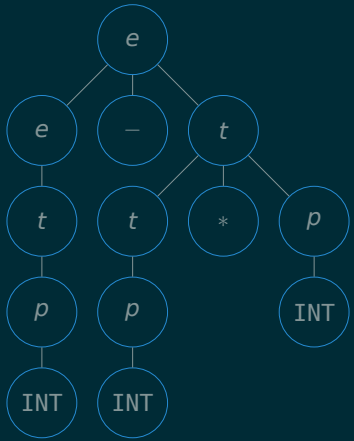
Parse the following input: `"1 - 2 * 3"`

We have the following terminals: `INT` $-$ `INT` $*$ `INT`

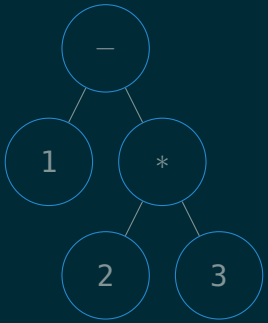
Previous Derivation

$$
\begin{aligned}
e &\Rightarrow e - t \\
&\Rightarrow t - t \\
&\Rightarrow p - t \\
&\Rightarrow \texttt{INT} - t \\
&\Rightarrow \texttt{INT} - t * p \\
&\Rightarrow \texttt{INT} - p * p \\
&\Rightarrow \texttt{INT} - \texttt{INT} * p \\
&\Rightarrow \texttt{INT} - \texttt{INT} * \texttt{INT}
\end{aligned}
$$

Previous Parse Tree



Previous AST



Our interpreter would return -5 which is correct

Grouping and Unary Precedence

Let's add grouping $()$ and a unary $-$

Grouping is always done at the highest level of precedence and calls the lowest level precedence rule

Unary operators follow the same rules as others, except there is no associativity (so it should call itself)

Our Completed Grammar

$$T = \{+, -, *, /, (,), \texttt{INT}\}$$
$$N = \{e, t, f, p\}$$
$$S = e$$
$$P = \begin{cases} e \to e + t \\ e \to e - t \\ e \to t \\ t \to t * f \\ t \to t / f \\ t \to f \\ f \to -f \\ f \to p \\ p \to (e) \\ p \to \texttt{INT} \end{cases}$$

Grammar Question

Consider the grammar:

$$
\begin{aligned}
T &= \{\texttt{IF}, \texttt{ELSE}\} \\
N &= \{\mathit{ifstmt}, \mathit{expr}, \mathit{stmt}\} \\
S &= \mathit{stmt} \\
P &= \begin{cases}
\mathit{stmt} \to \mathit{ifstmt} \\
\mathit{ifstmt} \to \texttt{IF}\ \mathit{expr}\ \mathit{stmt} \\
\mathit{ifstmt} \to \texttt{IF}\ \mathit{expr}\ \mathit{stmt}\ \texttt{ELSE}\ \mathit{stmt}
\end{cases}
\end{aligned}
$$

Is this ambiguous?

Grammar Answer

Yes, it is ambiguous

Consider the sentential form:
IF *expr* IF *expr stmt* ELSE *stmt*

We have two leftmost derivations of this form

Previous Derivations

$$
\begin{aligned}
\textit{stmt} &\Rightarrow \textit{ifstmt} \\
&\Rightarrow \texttt{IF}\ \textit{expr}\ \textit{stmt} \\
&\Rightarrow \texttt{IF}\ \textit{expr}\ \textit{ifstmt} \\
&\Rightarrow \texttt{IF}\ \textit{expr}\ \texttt{IF}\ \textit{expr}\ \textit{stmt}\ \texttt{ELSE}\ \textit{stmt}
\end{aligned}
$$

$$
\begin{aligned}
\textit{stmt} &\Rightarrow \textit{ifstmt} \\
&\Rightarrow \texttt{IF}\ \textit{expr}\ \textit{stmt}\ \texttt{ELSE}\ \textit{stmt} \\
&\Rightarrow \texttt{IF}\ \textit{expr}\ \textit{ifstmt}\ \texttt{ELSE}\ \textit{stmt} \\
&\Rightarrow \texttt{IF}\ \textit{expr}\ \texttt{IF}\ \textit{expr}\ \textit{stmt}\ \texttt{ELSE}\ \textit{stmt}
\end{aligned}
$$

This is called the dangling else problem

Dangling Else Example

Consider the following Java code:

```
if (x > 0) if (x > 3) x = 3 else x = 0
```

When `x = 1` the final value of x depends on our parse

Removing the Dangling Else

There are 4 main approaches:

- Add an `endif` statement (like in VHDL)
- Specify the meaning in the language specification and leave it up the implementations (like Java, which always does inner)
- Introduce indenting into the grammar (like Python)
- Require every `if` to have an `else` (like Haskell)

End Unambiguity

Start Techniques

Top-Down Parsing

Consider the grammar:

$$
\begin{aligned}
T &= \{0, 1\} \\
N &= \{s, b\} \\
S &= s \\
P &= \begin{cases} s \to 1b0 \\ s \to 0b1 \\ b \to 10 \\ b \to 11 \end{cases}
\end{aligned}
$$

Let's do our leftmost derivation one token at a time

This is called top-down parsing

Top-Down Parse (1)

Current token: 1

$$s \Rightarrow 1b0$$

Top-Down Parse (2)

Current token: 1

$$s \Rightarrow 1b0$$
$$s \Rightarrow 1100$$

This is a guess since we don't know the next token

Top-Down Parse (3)

Current token: 1

Our previous guess was wrong, so we backtrack to

$$s \Rightarrow 1b0$$

And guess again

$$s \Rightarrow 1b0$$
$$s \Rightarrow 1110$$

Top-Down Parse (4)

Current token: `0`

$$s \Rightarrow 1b0$$
$$s \Rightarrow 1110$$

This is the last token

Therefore `1110` is valid in our grammar

Recursive Descent Parsing

An example of a top-down parser is a recursive descent parser

Translating the grammar follows these rules:

- Each non-terminal turns into a function, the body turns into the production
- Each non-terminal in the production is a function call to the corresponding non-terminal
- Each terminal in the production is a call to the lexer's `consume` with the terminal as the argument

You can use `inspect` with a terminal as an argument to check the current token and decide which rule to follow

There could be backtracking as well if you need it

Implementing Our Grammar

Let's implement our first rule $e \rightarrow e + t$

One implementation is the following:

```
public void E() {
  E();
  consume("+");
  T();
}
```

Do you see the major problem with this?

Removing Left Recursion

Left recursion leads to infinite recursion in our function

We can remove left recursion in grammars of the form:

$$P = \begin{cases} a \to ax \\ a \to y \end{cases}$$

where a is a non-terminal, x and y are any sequence of terminals/non-terminals

By replacing it with:

$$P = \begin{cases} a \to ya' \\ a' \to xa' \\ a' \to \varepsilon \end{cases}$$

Note that ε represents no symbols

Removing Left Recursion Question

Remove left recursion from our rule:

$$P = \begin{cases} e \to e + t \\ e \to t \end{cases}$$

Removing Left Recursion Answer

Following the rules we would replace it with:

$$P = \begin{cases} e \to t\, e' \\ e' \to + t\, e' \\ e' \to \varepsilon \end{cases}$$

End Techniques

Start EBNF

Grammar Equivalence

Let's demonstrate that the grammars are equivalent:

$$P = \begin{cases} a \to ax \\ a \to y \end{cases}$$

$$P = \begin{cases} a \to ya' \\ a' \to xa' \\ a' \to \varepsilon \end{cases}$$

What are some example derivations?

Extended Backus-Naur Form

y, *yx*, *yxx*, *yxxx*

If *x* and *y* were just characters, this would be the same as the regular expression yx*

Extended Backus-Naur Form (EBNF) is just that, it's BNF with regular expression operations (+, *, ( ), ?)

EBNF Example

Assume the grammar:

$$
\begin{aligned}
T &= \{\mathtt{A}, \mathtt{B}, \mathtt{C}\} \\
N &= \{a, x\} \\
S &= a \\
P &= \Big\{ a \rightarrow \mathtt{A}\ x\ \mathtt{C}
\end{aligned}
$$

Let's consider some variations for the rule x in BNF and EBNF

EBNF Variation (1)

The following productions are equivalent:

$$P = \begin{cases} a \to \mathrm{A}\ x\ \mathrm{C} \\ x \to x\ \mathrm{B} \\ x \to \varepsilon \end{cases}$$

$$P = \begin{cases} a \to \mathrm{A}\ x\ \mathrm{C} \\ x \to \mathrm{B}* \end{cases}$$

$$P = \left\{ a \to \mathrm{A}\ \mathrm{B}*\ \mathrm{C} \right.$$

EBNF Variation (2)

The following productions are equivalent:

$$P = \begin{cases} a \to \mathrm{A}\ x\ \mathrm{C} \\ x \to \mathrm{B} \\ x \to \varepsilon \end{cases}$$

$$P = \begin{cases} a \to \mathrm{A}\ x\ \mathrm{C} \\ x \to \mathrm{B}? \end{cases}$$

EBNF Variation (3)

The following productions are equivalent:

$$P = \begin{cases} a \to \mathrm{A}\ x\ \mathrm{C} \\ x \to \mathrm{B}y \\ y \to \mathrm{B}y \\ y \to \varepsilon \end{cases}$$

$$P = \begin{cases} a \to \mathrm{A}\ x\ \mathrm{C} \\ x \to \mathrm{B}+ \end{cases}$$

EBNF Implementation Example

For recursive descent parsers $*$ translates to a `while` loop, $?$ translates to an `if` statement

$$P = \left\{ x \rightarrow \mathrm{B}* \right.$$

```
public void x() {
  while (inspect("B")) { consume("B"); }
}
```

$$P = \left\{ x \rightarrow \mathrm{B}? \right.$$

```
public void x() {
  if (inspect("B")) { consume("B"); }
}
```

EBNF Question

Consider our rewritten productions:

$$P = \begin{cases} e \to t\, e' \\ e' \to + t\, e' \\ e' \to \varepsilon \end{cases}$$

Let's convert this grammar to EBNF

EBNF Answer

Recognize that e' can be transformed:

$$P = \left\{ e \rightarrow t \ (+ \ t) * \right.$$

Converting Our Grammar to EBNF

We also have a $-$ at that level of precedence:

$$P = \begin{cases} e \to t\ (+\ t)* \\ e \to t\ (-\ t)* \end{cases}$$

We can factor out the common parts of both productions

$$P = \Big\{ e \to t\ ((+\ t)\ |\ (-\ t))*$$

$$P = \Big\{ e \to t\ ((+|-)\ t)*$$

The production is now in a form that's easily translated into a recursive descent parser

EBNF for Associativity

Let's specify our previous associativity rules in EBNF

Consider e as the current level of precedence and p is the next highest level of precedence

$e \rightarrow p\ (\mathsf{OP}\ p)*$ is left associative

$e \rightarrow p\ (\mathsf{OP}\ e)?$ is right associative

End EBNF

Start Language Theory

Deterministic Top-Down Parsing

We can have a deterministic recursive descent parser if we have no backtracking

To avoid backtracking, we make only correct guesses

To make only correct guesses, we restrict the grammar so we can always choose the correct next step based on the next token

LL(1)

LL is the class of grammars you've been parsing

- L for left-to-right scan of the input
- L for leftmost derivation

An $LL(k)$ grammar can be parsed top-down with no backtracking looking only at the next k symbols in the input

$LL(1)$ can be parsed with a recursive descent parser with a one token (or terminal) look-ahead

End Language Theory