Lexing and Parsing

Plan for this week

Story $\xrightarrow{\text{"parse"}}$ $\boxed{\text{EXPR}}$ $\xrightarrow{\text{"eval"}}$ VALUE

Type check

Last week:

- How do we *evaluate* a program given its AST?

```
eval :: Env -> Expr -> Value
```

This week:

- How do we *convert* program text into an AST?

```
parse :: String -> Expr
```

Example: calculator with variables

AST representation:

```
data Aexpr
  = AConst  Int
  | AVar    Id
  | APlus   Aexpr Aexpr
  | AMinus  Aexpr Aexpr
  | AMul    Aexpr Aexpr
  | ADiv    Aexpr Aexpr
```

Evaluator:

```
eval :: Env -> Aexpr -> Value
...
```

Using the evaluator:

```
λ> eval [] (APlus (AConst 2) (AConst 6))
8

λ> eval [("x", 16), ("y", 10)] (AMinus (AVar "x") (AVar "y"))
6

λ> eval [("x", 16), ("y", 10)] (AMinus (AVar "x") (AVar "z"))
*** Exception: Error {errMsg = "Unbound variable z"}
```

But writing ASTs explicitly is really tedious, we are used to writing programs as text!

We want to write a function that converts strings to ASTs if possible:

```
parse :: String -> Aexpr
```

For example:

```
λ> parse "2 + 6"
APlus (AConst 2) (AConst 6)

λ> parse "(x - y) / 2"
ADiv (AMinus (AVar "x") (AVar "y")) (AConst 2)

λ> parse "2 +"
*** Exception: Error {errMsg = "Syntax error"}
```

Two-step-strategy

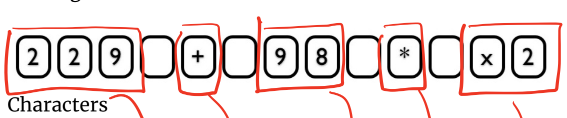
How do I read a sentence "He ate a bagel"?

- First split into words: `["He", "ate", "a", "bagel"]`
- Then relate words to each other: "He" is the subject, "ate" is the verb, etc

Let's do the same thing to "read" programs!

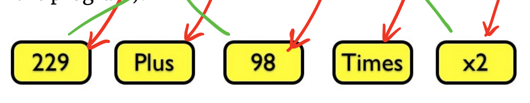
Step 1 (Lexing) : From String to Tokens

A string is a list of *characters*:



Characters

First we aggregate characters that "belong together" into **tokens** (i.e. the "words" of the program):



Tokens

We distinguish tokens of different kinds based on their format:

- all numbers: integer constant
- alphanumeric, starts with a letter: identifier
- `+` : plus operator
- etc

Step 2 (Parsing) : From Tokens to AST

Next, we convert a sequence of tokens into an AST

- This is hard...
- ... but the hard parts do not depend on the language!

Parser generators

- Given the description of the *token format* generates a *lexer*
- Given the description of the *grammar* generates a *parser*

We will be using parser generators, so we only care about how to describe the token format and the grammar

lex   yacc

yet another compiler compiler

Lexing

HAPPY

We will use the tool called `alex` to generate the **lexer**

Input to `alex` : a `.x` file that describes the *token format*

Tokens

① Lexing

First we list the kinds of tokens we have in the language:

```
data Token
  = NUM    AlexPosn Int
  | ID     AlexPosn String
  | PLUS   AlexPosn
  | MINUS  AlexPosn
  | MUL    AlexPosn
  | DIV    AlexPosn
  | LPAREN AlexPosn
  | RPAREN AlexPosn
  | EOF    AlexPosn
```

Token rules

r+

Next we describe the format of each kind of token using a rule:

REGEXP

```
[\+]                          { \p _ -> PLUS   p }
[\-]                          { \p _ -> MINUS  p }
[\*]                          { \p _ -> MUL    p }
[\/]                          { \p _ -> DIV    p }
\(                            { \p _ -> LPAREN p }
\)                            { \p _ -> RPAREN p }
$alpha [$alpha $digit \_ \']* { \p s -> ID     p s }
$digit+                       { \p s -> NUM p (read s) }
```

Each line consist of:

- a *regular expression* that describes which strings should be recognized as this token
- a Haskell expression that generates the token

You read it as:

- if at position `p` in the input string
- you encounter a substring `s` that matches the *regular expression*
- evaluate the Haskell expression with arguments `p` and `s`

Regular Expressions

A regular expression has one of the following forms:

- `[c1 c2 ... cn]` matches *any of* the characters `c1 .. cn`
  - `[0-9]` matches *any digit*
  - `[a-z]` matches *any lower-case letter*
  - `[A-Z]` matches *any upper-case letter*
  - `[a-z A-Z]` matches *any letter*
- `R1 R2` matches a string `s1 ++ s2` where `s1` matches `R1` and `s2` matches `R2`
  - e.g. `[0-9] [0-9]` matches any two-digit string
- `R+` matches *one or more* repetitions of what `R` matches
  - e.g. `[0-9]+` matches a natural number
- `R*` matches *zero or more* repetitions of what `R` matches

QUIZ

Which of the following strings are matched by `[a-z A-Z] [a-z A-Z 0-9]*`?

begins with "letter"

(A) (empty string) ✗

(B) `5` ✗

(C) `x5` ✓

(D) `x` ✓

(E) C and D ✓

x_5

Back to token rules

We can **name** some common regexps like:

```
$digit = [0-9]
$alpha = [a-z A-Z]
```

and write `[a-z A-Z] [a-z A-Z 0-9]*` as `$alpha [$alpha $digit]*`

```
[\+]                           { \p _ -> PLUS   p }
[\-]                           { \p _ -> MINUS  p }
[\*]                           { \p _ -> MUL    p }
[\/]                           { \p _ -> DIV    p }
\(                             { \p _ -> LPAREN p }
\)                             { \p _ -> RPAREN p }
$alpha [$alpha $digit \_ \']*  { \p s -> ID     p s }
$digit+                        { \p s -> NUM p (read s) }
```

- When you encounter a `+` , generate a `PLUS` token
- ...
- When you encounter a nonempty string of digits, convert it into an integer and generate a `NUM`
- When you encounter an alphanumeric string that starts with a letter, save it in an`ID token

Running the Lexer

From the token rules, `alex` generates a function `alexScan` which

- given an input string, find the *longest* prefix `p` that matches one of the rules
- if `p` is empty, it fails
- otherwise, it converts `p` into a token and returns the rest of the string

We wrap this function into a handy function

```
parseTokens :: String -> Either ErrMsg [Token]
```

which repeatedly calls `alexScan` until it consumes the whole input string or fails

We can test the function like so:

```
λ> parseTokens "23 + 4 / off -"
Right [ NUM (AlexPn 0 1 1) 23
      , PLUS (AlexPn 3 1 4)
      , NUM (AlexPn 5 1 6) 4
      , DIV (AlexPn 7 1 8)
      , ID (AlexPn 9 1 10) "off"
      , MINUS (AlexPn 13 1 14)
      ]

λ> parseTokens "%"
Left "lexical error at 1 line, 1 column"
```

QUIZ

What is the result of `parseTokens "92zoo"` (positions omitted for readability)?

(A) Lexical error

(B) `[ID "92zoo"]`

(C) `[NUM "92"]`

(D) `[NUM "92", ID "zoo"]` ✓

Parsing

We will use the tool called `happy` to generate the **parser**

Input to `happy` : a `.y` file that describes the *grammar*

Wait, wasn't this the grammar?

```
data Aexpr
  = AConst  Int
  | AVar    Id
  | APlus   Aexpr Aexpr
  | AMinus  Aexpr Aexpr
  | AMul    Aexpr Aexpr
  | ADiv    Aexpr Aexpr
```

This was *abstract syntax*

Now we need to describe *concrete syntax*

- What programs look like when written as text
- and how to map that text into the abstract syntax

Grammars

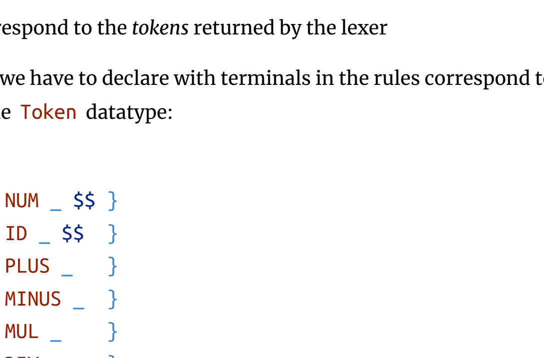
A grammar is a recursive definition of a set of trees

- each tree is a *parse tree* for some string
- *parse* a string `s` = find a parse tree for `s` that belongs to the grammar

A grammar is made of:

- **Terminals**: the leaves of the tree (tokens!)
- **Nonterminals:** the internal nodes of the tree
- **Production Rules** that describe how to "produce" a non-terminal from terminals and other non-terminals
  - i.e. what children each nonterminal can have:

```
Aexpr :   -- NT Aexpr can have as children:
1 | Aexpr '+' Aexpr  { ... } -- NT Aexpr, T '+', and NT Aexpr, or
2 | Aexpr '-' AExpr  { ... } -- NT Aexpr, T '-', and NT Aexpr, or
3 | ...
4  Num
```



Terminals

Terminals correspond to the *tokens* returned by the lexer

In the `.y` file, we have to declare with terminals in the rules correspond to which tokens from the `Token` datatype:

```
%token
    TNUM  { NUM _ $$ }
    ID    { ID _ $$  }
    '+'   { PLUS _   }
    '-'   { MINUS _  }
    '*'   { MUL _    }
    '/'   { DIV _    }
    '('   { LPAREN _ }
    ')'   { RPAREN _ }
```

- Each thing on the left is terminal (as appears in the production rules)
- Each thing on the right is a Haskell pattern for datatype `Token`
- We use `$$` to designate one parameter of a token constructor as the **value** of that token
  - we will refer back to it from the production rules

Production rules

Next we define productions for our language:

```
Aexpr : TNUM                 { AConst $1    }
      | ID                   { AVar   $1    }
      | '(' Aexpr ')'        { $2           }
      | Aexpr '*' Aexpr      { AMul   $1 $3 }
      | Aexpr '+' Aexpr      { APlus  $1 $3 }
      | Aexpr '-' Aexpr      { AMinus $1 $3 }
```

The expression on the right computes the *value* of this node

- `$1 $2 $3` refer to the *values* of the respective child nodes

Example: parsing `(2)` as `AExpr` :

1. Lexer returns a sequence of `Token` s: `[LPAREN, NUM 2, RPAREN]`
2. `LPAREN` is the token for terminal `'('` , so let's pick production `'(' Aexpr ')'`
3. Now we have to parse `NUM 2` as `Aexpr` and `RPAREN` as `')'`
4. `NUM 2` is a token for nonterminal `TNUM` , so let's pick production `TNUM`
5. The value of this `Aexpr` node is `AConst 2` , since the value of `TNUM` is `2`
6. The value of the top-level `Aexpr` node is also `AConst 2` (see the `'(' Aexpr ')'` production)

QUIZ

What is the value of the root `AExpr` node when parsing `1 + 2 + 3`?

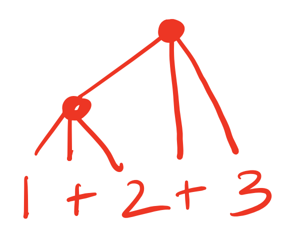
```
Aexpr : TNUM                 { AConst $1    }
```

```
Aexpr : TNUM                    { AConst $1     }
      | ID                      { AVar   $1     }
      | '(' Aexpr ')'           { $2            }
      | Aexpr '*' Aexpr         { AMul   $1 $3  }
      | Aexpr '+' Aexpr         { APlus  $1 $3  }
      | Aexpr '-' Aexpr         { AMinus $1 $3  }
```

(A) Cannot be parsed as `AExpr` ✗

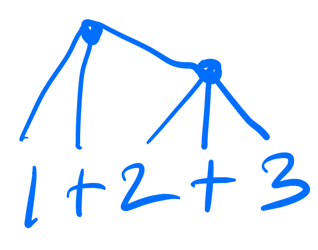
(B) `6` ✗

(C) `APlus (APlus (AConst 1) (AConst 2)) (AConst 3)`



(D) `APlus (AConst 1) (APlus (AConst 2) (AConst 3))`

((((1 + (2) + (3) + (4) + 5))))



Running the Parser

First, we should tell the parser that the top-level non-terminal is `AExpr`:

```
%name aexpr
```

From the production rules and this line, `happy` generates a function `aexpr` that tries to parse a sequence of tokens as `AExpr`

We package this function together with the lexer and the evaluator into a handy function

```
evalString :: Env -> String -> Int
```

We can test the function like so:

```
λ> evalString [] "1 + 3 + 6"
10

λ> evalString [("x", 100), ("y", 20)] "x - y"
???

λ> evalString [] "2 * 5 + 5"
???

λ> evalString [] "2 - 1 - 1"
???
```

Precedence and associativity

```
λ> evalString [] "2 * 5 + 5"
20
```

The problem is that our grammar is **ambiguous**!

There are multiple ways of parsing the string `2 * 5 + 5`, namely

- `APlus (AMul (AConst 2) (AConst 5)) (AConst 5)` (good)
- `AMul (AConst 2) (APlus (AConst 5) (AConst 5))` (bad!)

Wanted: tell `happy` that `*` has **higher precedence** than `+`!

```
λ> evalString [] "2 - 1 - 1"
2
```

There are multiple ways of parsing `2 - 1 - 1`, namely

- `AMinus (AMinus (AConst 2) (AConst 1)) (AConst 1)` (good)
- `AMinus (AConst 2) (AMinus (AConst 1) (AConst 1))` (bad!)

Wanted: tell `happy` that `-` is **left-associative**!

How do we communicate precedence and associativity to `happy`?

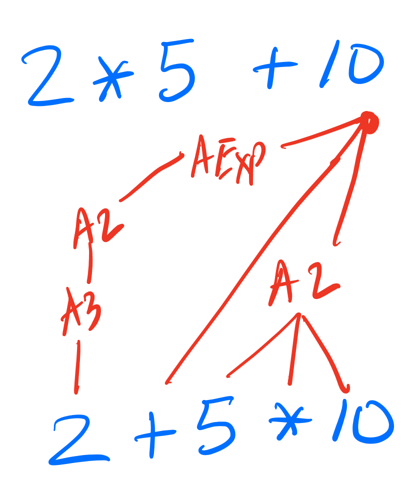
Solution 1: Grammar factoring

We can split the `AExpr` non-terminal into multiple "levels"

```
Aexpr : Aexpr '+' Aexpr2
      | Aexpr '-' Aexpr2
      | Aexpr2

Aexpr2 : Aexpr2 '*' Aexpr3
       | Aexpr2 '/' Aexpr3
       | Aexpr3

Aexpr3 : TNUM
       | ID
       | '(' Aexpr ')'
```



Intuition: `AExpr2` "binds tighter" than `AExpr`, and `AExpr3` is the tightest

Now I cannot parse the string `2 * 5 + 5` as

- `AMul (AConst 2) (APlus (AConst 5) (AConst 5))`
- Why?

Solution 2: Parser directives

This problem is so common that parser generators have a special syntax for it!

```
%left '+' '-'
%left '*' '/'
```

What this means:

- All our operators are left-associative
- Operators on the lower line have higher precedence

((12 - 2) - 2)

(f (10 (20 30)))

That's all folks!