Compilers and Formal Languages

Email: christian.urban at kcl.ac.uk

Slides & Progs: KEATS (also homework is there)

1 Introduction, Languages	6 While-Language
2 Regular Expressions, Derivatives	7 Compilation, JVM
3 Automata, Regular Languages	8 Compiling Functional Languages
4 Lexing, Tokenising	9 Optimisations
5 Grammars, Parsing	10 LLVM

Parser Combinators

One of the simplest ways to implement a parser, see `https://vimeo.com/142341803` (by Haoyi Li)

- build-in library in Scala
- fastparse (2) library by Haoyi Li; is part of Ammonite
- possible exponential runtime behaviour

Parser Combinators

Parser combinators:

$$\underbrace{\text{list of tokens}}_{\text{input}} \Rightarrow \underbrace{\text{set of (parsed input, unparsed input)}}_{\text{output}}$$

- atomic parsers
- sequencing
- alternative
- semantic action (map-parser)

Atomic parsers, for example, number tokens

$$\texttt{Num(123)} :: \mathit{rest} \;\Rightarrow\; \{(\texttt{Num(123)}, \mathit{rest})\}$$

you consume one or more token from the input (stream)

also works for characters and strings

Alternative parser (code $p \mid\mid q$)

apply p and also q; then combine the outputs

$$p(\text{input}) \cup q(\text{input})$$

Sequence parser (code $p \sim q$)

apply first p producing a set of pairs
then apply q to the unparsed parts
then combine the results:

$$((\text{output}_1, \text{output}_2), \text{unparsed part})$$

$$\{((o_1, o_2), u_2) \mid (o_1, u_1) \in p(\text{input}) \wedge (o_2, u_2) \in q(u_1)\}$$

Map-parser (code $p.map(f)$)

apply p producing a set of pairs

then apply the function f to each first component

$$\{(f(o_1), u_1) \mid (o_1, u_1) \in p(\text{input})\}$$

Map-parser (code $p.map(f)$)

apply p producing a set of pairs

then apply the function f to each first component

$$\{(f(o_1), u_1) \mid (o_1, u_1) \in p(\text{input})\}$$

f is the semantic action ("what to do with the parsed input")

Semantic Actions

Addition

$$T \sim + \sim E \Rightarrow \underbrace{f((x,y),z) \Rightarrow x + z}_{\text{semantic action}}$$

Semantic Actions

Addition

$$T \sim + \sim E \Rightarrow \underbrace{f((x,y),z) \Rightarrow x + z}_{\text{semantic action}}$$

Multiplication

$$F \sim * \sim T \Rightarrow f((x,y),z) \Rightarrow x * z$$

Semantic Actions

Addition

$$T \sim + \sim E \Rightarrow \underbrace{f((x,y),z) \Rightarrow x + z}_{\text{semantic action}}$$

Multiplication

$$F \sim * \sim T \Rightarrow f((x,y),z) \Rightarrow x * z$$

Parenthesis

$$(\sim E \sim) \Rightarrow f((x,y),z) \Rightarrow y$$

Input Types of Parsers

input: token list

output: set of (output_type, token list)

Input Types of Parsers

input: token list

output: set of (output_type, token list)

actually it can be any input type as long as it is a kind of sequence (for example a string)

Scannerless Parsers

input: string

output: set of (output_type, string)

but using lexers is better because whitespaces or comments can be filtered out; then input is a sequence of tokens

Abstract Parser Class

```
abstract class Parser[I, T] {
  def parse(ts: I): Set[(T, I)]

  def parse_all(ts: I) : Set[T] =
    for ((head, tail) <- parse(ts);
         if (tail.isEmpty)) yield head
}
```

Successful Parses

input: string

output: set of (output_type, string)

a parse is successful whenever the input has been fully "consumed" (that is the second component is empty)

```
start := 1000;
x := start;
y := start;
z := start;
while 0 < x do {
 while 0 < y do {
  while 0 < z do { z := z - 1 };
  z := start;
  y := y - 1
 };
 y := start;
 x := x - 1
}
```

While-Language

$$
\begin{array}{rcl}
\textit{Stmt} & ::= & \text{skip} \\
& | & \textit{Id} := \textit{AExp} \\
& | & \text{if } \textit{BExp} \text{ then } \textit{Block} \text{ else } \textit{Block} \\
& | & \text{while } \textit{BExp} \text{ do } \textit{Block} \\
\textit{Stmts} & ::= & \textit{Stmt} \,;\, \textit{Stmts} \\
& | & \textit{Stmt} \\
\textit{Block} & ::= & \{\, \textit{Stmts} \,\} \\
& | & \textit{Stmt} \\
\textit{AExp} & ::= & \ldots \\
\textit{BExp} & ::= & \ldots
\end{array}
$$

Aexps

$$
\begin{array}{lcl}
\text{eval}(n) & \stackrel{\text{def}}{=} & n \\
\text{eval}(a_1 + a_2) & \stackrel{\text{def}}{=} & \text{eval}(a_1) + \text{eval}(a_2) \\
\text{eval}(a_1 - a_2) & \stackrel{\text{def}}{=} & \text{eval}(a_1) - \text{eval}(a_2) \\
\text{eval}(a_1 * a_2) & \stackrel{\text{def}}{=} & \text{eval}(a_1) * \text{eval}(a_2) \\
\\
\text{eval}(x) & \stackrel{\text{def}}{=} & ???
\end{array}
$$

Interpreter

$$
\begin{array}{lcll}
\mathrm{eval}(n, E) & \stackrel{\text{def}}{=} & n & \\
\mathrm{eval}(x, E) & \stackrel{\text{def}}{=} & E(x) & \text{lookup } x \text{ in } E \\
\mathrm{eval}(a_1 + a_2, E) & \stackrel{\text{def}}{=} & \mathrm{eval}(a_1, E) + \mathrm{eval}(a_2, E) & \\
\mathrm{eval}(a_1 - a_2, E) & \stackrel{\text{def}}{=} & \mathrm{eval}(a_1, E) - \mathrm{eval}(a_2, E) & \\
\mathrm{eval}(a_1 * a_2, E) & \stackrel{\text{def}}{=} & \mathrm{eval}(a_1, E) * \mathrm{eval}(a_2, E) & \\
 & & & \\
\mathrm{eval}(a_1 = a_2, E) & \stackrel{\text{def}}{=} & \mathrm{eval}(a_1, E) = \mathrm{eval}(a_2, E) & \\
\mathrm{eval}(a_1 != a_2, E) & \stackrel{\text{def}}{=} & \neg(\mathrm{eval}(a_1, E) = \mathrm{eval}(a_2, E)) & \\
\mathrm{eval}(a_1 < a_2, E) & \stackrel{\text{def}}{=} & \mathrm{eval}(a_1, E) < \mathrm{eval}(a_2, E) &
\end{array}
$$

An Interpreter (1)

$$
\begin{array}{l}
\{ \\
\quad x := 5; \\
\quad y := x * 3; \\
\quad y := x * 4; \\
\quad x := u * 3 \\
\}
\end{array}
$$

the interpreter has to record the value of x before assigning a value to y

An Interpreter (1)

$$
\begin{array}{l}
\{ \\
\quad x := 5; \\
\quad y := x * 3; \\
\quad y := x * 4; \\
\quad x := u * 3 \\
\}
\end{array}
$$

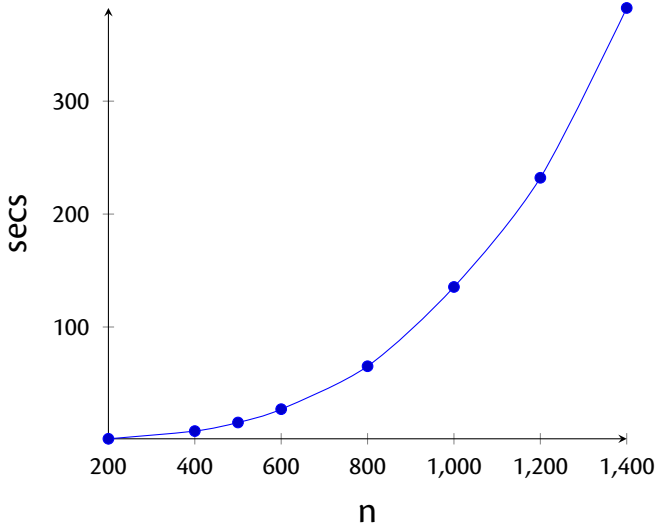
the interpreter has to record the value of x before assigning a value to y

`eval(stmt, env)`

Interpreter (2)

$$\begin{array}{lcl}
\text{eval}(\text{skip}, E) & \stackrel{\text{def}}{=} & E \\
\text{eval}(x := a, E) & \stackrel{\text{def}}{=} & E(x \mapsto \text{eval}(a, E)) \\
\text{eval}(\text{if } b \text{ then } cs_1 \text{ else } cs_2, E) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{l}{\qquad \text{if eval}(b, E) \text{ then eval}(cs_1, E)} \\
\multicolumn{3}{l}{\qquad\qquad\qquad \text{else eval}(cs_2, E)} \\
\text{eval}(\text{while } b \text{ do } cs, E) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{l}{\qquad \text{if eval}(b, E)} \\
\multicolumn{3}{l}{\qquad \text{then eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E))} \\
\multicolumn{3}{l}{\qquad \text{else } E} \\
\text{eval}(\text{write } x, E) & \stackrel{\text{def}}{=} & \{\ \text{println}(E(x))\ ;\ E\ \}
\end{array}$$

Interpreted Code



In CW3, in the collatz program there is the line write "\n" Should this print "/n" or perform the new line command /n ? Also should write be print() or println() ?