Verified Parsing

Michael Norrish
Canberra Research Lab., NICTA

Tuesday, 19 November 2013

Outline

Specifying a Syntax

Implementing a Syntax
- Parsing Expression Grammars (PEGs)
- Verifying a PEG

Verification needs Specification

If we are to **verify** anything, we (implicitly?) have some expectation that the thing-to-be-verified can fail to achieve.

Turning **expectation** into **specification** can be quite tricky.

Expectation *vs* Specification

Consider verifying

- a user interface
- an operating system
- a word-processor

Bugs might be easy to spot in all of the above; but it may also be quite hard to write down the formal statement of what is wanted.

Specifications for Language Syntax

Luckily, we have known since the 1960s that language syntax can be specified with **context free grammars**.

- Many textbooks have been written…
- Many undergraduates have been taught…

Noam Chomsky has a lot to answer for.

The Standard Story (c1975)

1. Specify lexical syntax with regular expressions
   - (implement with DFA construction (*e.g.*, `lex`))
2. Specify concrete syntax with CFGs
   - (implement with LR-parsing (*e.g.*, `yacc`))
3. Use "parser actions" to turn parse-trees into abstract syntax
4. Type-checking, code generation, *etc.*, *etc.*

[See, for example, the famous "Dragon Book".]

The Parsing Problem

Given a CFG G and an input string s return a parse-tree t such that

- t conforms to the rules of grammar G,
- G's start symbol is the root symbol of t, and
- the fringe of t is equal to all of s

We're already assuming that G is appropriate, suitably unambiguous *etc*.

In practice, t may be transformed as it is generated.

We've All Drunk the CFG Kool-Aid

Stuff like

```
selection-statement:
    if  ( condition ) statement
    if  ( condition ) statement else statement
    switch ( condition ) statement
```

can come to seem quite natural.

CFGs Are Not Without Their Flaws

Getting precedence right ties a grammar into verbose knots.

Many “perfectly reasonable” things are impossible to get right (examples to come).

Implementing them is still not quite a solved problem.

Classic CFG Problems: Dangling `else`

Our friend

```
selection-statement:
    if  ( condition ) statement
    if  ( condition ) statement else statement
    switch ( condition ) statement
```

is ambiguous.

There are two different parse-trees for

```
if (x) if (y) x++; else x--;
```

Classic CFG Problems: C's `typedef`

Declare `x` to be a pointer to an array of three integers, and initialise it with value `v`:

```
typedef int f;
f(*x)[3] = v;
```

Dereference `x`, pass it to function `f`, index into the returned pointer at position 3, and assign `v` there.

```
/* typedef int f; */
f(*x)[3] = v;
```

Advantages to CFGs

Familiar.

- Maybe even “natural”.

Well-developed theory.

- See, for example, Hopcroft and Ullman.

Well-developed tools.

No obvious alternative.

Outline

Specifying a Syntax

Implementing a Syntax
- Parsing Expression Grammars (PEGs)
- Verifying a PEG

What Not to Do

1. Write custom parser in C
2. Debug by checking it on 2 sample programs
3. Start to verify it

Verifying the `yacc` "Compiler"

`yacc` is the classic parser-generator program.

- "Yet Another Compiler Compiler"

It compiles a source program (specification), the CFG.

And produces a program for parsing with respect to that grammar.

Verified *vs* Verifying (Digression)

When verifying a compiler, there are two well-known approaches:

Verified Compiler

Verify (once and for all) that compiler source code will always transform source code into equivalent object code.

Example: CompCert, CakeML.

Verified *vs* Verifying (Digression)

When verifying a compiler, there are two well-known approaches:

Verifying Compiler

For each run of compiler, independently confirm that object code and source code are equivalent.

Example: seL4 C.

Verified `yacc`

Work with PhD student Aditi Barthwal (2009).

- Theory of CFGs in HOL4
- Formal presentation of SLR automaton construction
- Proofs that resulting automaton is
  - **sound:** if the automaton accepts the string, it is in the language of the grammar
  - **complete:** if a string is in the language, the automaton will accept it

Verified `yacc`'s Omissions

`yacc` is actually LALR, not SLR.

- Probably not a big job to make this extension

Compiler is entirely within the logic; the object code is not C, or SML, or ...

No proof of termination.

- If a string is not in the language, the automaton is not proven to terminate

Verifying `yacc`

Work by Jourdan, Pottier, and Leroy (2012).

- Verifies that an LALR automaton will correctly parse a CFG's language
- Used in the CompCert compiler (addressing bugs)
  - Note beautiful way in which a *verified* compiler includes a *verifying* component
- No termination proof

Everyone Hates `yacc`

...

Also, dealing with automata is painful if you are targetting a language without arrays.

The Seductive World of LL Parsing

Take

```
selection-statement:
      if  ( condition ) statement
      if  ( condition ) statement else statement
      switch ( condition ) statement
```

Write

```
function parse-selection =
  case next-token of
    TOK If =>
      grab(TOK Lparen);
      parse-condition;
      grab(TOK Rparen);
      ...
  | TOK Switch => ...
  | _ => raise Error
```

The Seductive World of LL Parsing

Write code that mimicks the structure of the input grammar.

- Calculate (by hand) things like *first* sets

This starts to feel as if it's almost ... mechanical.

But verification is still hard because it's of source code in a full programming language.

Enter the Domain Specific Language

Programming in a well-specified subset.

Thus:

- **Monadic parser combinators** allow a very appealing combination

```
parse-selection =
    (grab(TOK If) >>
     grab(TOK LParen) >>
     ...) ||
    (grab(TOK Switch) >>
     ...)
```

- Code may still be laced with arbitrary Haskell (say)

Parsing Expression Grammars (PEGs)

Another DSL (Ford 2004).

$$\begin{aligned} e \;::=\; & \varepsilon \mid . \mid c \mid e_1\, e_2 \mid e^* \\ & \mid\; e_1 / e_2 \mid \neg e \mid N \end{aligned}$$

where N is the name of a non-terminal symbol.

PEGs Informally

ε	**Empty**. Consume no input, succeed.
.	**Any**. Consume one token from input. Fail if at end of input
c	**Char**. Consume c from input. Fail if it's not there, or if at end.
$e_1 e_2$	**Sequence**. Run e_1 then e_2. Fail if either fails.
e^*	**Repeat**. Run e repeatedly until it fails. Always succeeds.
e_1 / e_2	**Choice**. Try e_1. If it fails, try e_2. Fails if both fail.
$\neg e$	**Not**. Try e. Fail if it succeeds, succeed if it fails.
N	**Non-Terminal**. Look up N, run its expression.

PEG Derived Forms

$e^?$	(e/ε)	**Option**.
e^+	(ee^*)	**Repeat One**.
$[a{-}z]$	$(a/b/\cdots/z)$	**Range**.
$\&e$	$(\neg\neg e)$	**Lookahead**.

PEGs are a Nice DSL

Can capture many typical idioms.

Short-circuiting choice, repeat and not are powerful features.

Choice is **deterministic**, so PEGs are too.

E.g.

```
If ( Condition ) Statement (else Statement)?  /
Switch ( Condition ) Statement
```

gives the correct “else attaches to nearest if” behaviour.

PEG Choice Gives Backtracking

Something like

$$e_1\, e_2 \;/\; e_3$$

may do large amounts of work on e_1 and/or e_2 before ultimately giving up and going to e_3.

With great power comes great responsibility.

Recursion

Repetition (e^*) covers many instances.

If the grammar calls for:

$$E ::= E + T \mid T$$

$$T ::= n \mid (E)$$

Can replace with

$$E \mapsto (T +)^* T$$

$$T \mapsto n \;/\; (E)$$

(Note $T \to E \to T$ recursion is still present in both.)

PEG Grammar Engineering

Think about back-tracking in

$$\begin{aligned} E &\mapsto (T\,\mathbf{+})^*\,T \\ T &\mapsto n \;/\; (E) \end{aligned}$$

and you may want to write

$$\begin{aligned} E &\mapsto T\,(\mathbf{+}\,T)^* \\ T &\mapsto n \;/\; (E) \end{aligned}$$

instead.

PEGs Aren't All Well-formed

We can't write the direct analogue of the input CFG:

$$E \mapsto E + T \;/\; T$$
$$T \mapsto n \;/\; (E)$$

Though we could (more engineering!) write

$$E \mapsto T + E \;/\; T$$
$$T \mapsto n \;/\; (E)$$

PEGs are an "LL technology" and left-recursion can't be used.

Formalising PEGs

Define an inductive relation

$$(e, s_1) \rightsquigarrow \checkmark_{s_2} \qquad\qquad (e, s) \rightsquigarrow \bot$$

Two cases:

- PEG e starts on input s_1 and successfully consumes it, leaving s_2 as remaining input; or
- PEG e starts on input s and fails.

PEG Rules

Some simple cases:

$$\frac{}{(\varepsilon, s) \rightsquigarrow \checkmark_s} \qquad \frac{N \mapsto e \quad (e, s) \rightsquigarrow r}{(N, s) \rightsquigarrow r}$$

$$\frac{}{(c, c :: s) \rightsquigarrow \checkmark_s} \qquad \frac{c_1 \neq c_2}{(c_1, c_2 :: s) \rightsquigarrow \bot}$$

$$\frac{}{(., c :: s) \rightsquigarrow \checkmark_s} \qquad \frac{}{(., []) \rightsquigarrow \bot}$$

$$\frac{(e_1, s_0) \rightsquigarrow \checkmark_{s_1} \quad (e_2, s_1) \rightsquigarrow \checkmark_s}{(e_1 e_2, s_0) \rightsquigarrow \checkmark_s}$$

More Interesting PEG Rules

$$\frac{(e_1, s_0) \rightsquigarrow \checkmark_s}{(e_1/e_2, s_0) \rightsquigarrow \checkmark_s}$$

$$\frac{(e_1, s) \rightsquigarrow \bot \quad (e_2, s) \rightsquigarrow r}{(e_1/e_2, s) \rightsquigarrow r}$$

$$\frac{(e, s) \rightsquigarrow \bot}{(\neg e, s) \rightsquigarrow \checkmark_s}$$

$$\frac{(e, s) \rightsquigarrow \checkmark_{s'}}{(\neg e, s) \rightsquigarrow \bot}$$

$$\frac{(e, s) \rightsquigarrow \bot}{(e^*, s) \rightsquigarrow \checkmark_s}$$

$$\frac{(e, s_0) \rightsquigarrow \checkmark_{s_1} \quad (e^*, s_1) \rightsquigarrow \checkmark_s}{(e^*, s_0) \rightsquigarrow \checkmark_s}$$

Theorems About PEGs

Deterministic. Order of evaluation in things like choice and sequencing is completely specified.

Total on Good PEGs. If a grammar doesn't have left-recursion, the relation will define a result for all inputs.

Adding Semantic Actions

As defined, PEGs just give yes/no verdicts on string acceptability.

To be **practical**, they should give the user access to the (implicit) parse-tree.

Associate each expression in a grammar with a function that gets passed the results of recursive calls.

Semantic Actions

- Associate a function f with e^*, such that f gets passed a list of values generated from the recursive calls to e.
  Then, f must return a value that will be used in turn by the levels above.
- Associate a function g with $e_1 e_2$, such that g gets passed two values, one from e_1 and one from e_2.
- Associate a value v with $\neg e$, which is returned when e fails.

Executing PEGs

PEGs have to be presented as relations because of the possibility of **non-termination**.

However, they have an obvious interpretation in your favourite programming language.

The "PEG Interpreter"

- might loop if the input grammar is malformed;
- but will return the right result otherwise.

Verifying a PEG

In the CakeML project, we had a CFG for our SML subset.

We wrote a PEG to parse token streams, and to return the **concrete syntax tree**.

The PEG's semantic actions explicitly built that tree.

- all interesting semantic analysis came later

But Wait!

PEGs are written in high-level domain-specific language.

Perhaps the PEG is just an **executable specification** in itself.

- So clear that it's obviously what we **expected**.

And that would mean nothing to verify :-)

CakeML PEG as Executable Spec

For example:

```
mkNT nEbase ↦
  choicel [tok isInt (bindNT nEbase o mktokLf);
           seql [tokeq LparT; tokeq RparT]
                (bindNT nEbase);
           peg_EbaseParen;
           seql [tokeq LbrackT; try (pnt nElist1);
                 tokeq RbrackT]
                (bindNT nEbase);
           seql [tokeq LetT; pnt nLetDecs; tokeq InT;
                 pnt nEseq; tokeq EndT]
                (bindNT nEbase);
           pegf (pnt nFQV) (bindNT nEbase);
           pegf (pnt nConstructorName)
                (bindNT nEbase)]
```

CakeML PEG as Executable Spec

For left-associating expressions:

```
mkNT nEmult ↦
  peg_linfix (mkNT nEmult) (pnt nEapp) (pnt nMultOps)
```

Like:

```
  nEmult ::= nEmult nMultOps nEapp | nEapp
```

But

```
  peg_linfix tgtnt rptsym opsym =
    seq rptsym (rpt (seq opsym rptsym (++)) FLAT)
        (λa b.
          case a of
            [] => []
          | h::_ => [mk_linfix
                       tgtnt
                       (Nd tgtnt [h])
                       b])
```

What Are We Trying to Achieve?

Let's get all existential.

We want to create trust in our system.

If we had no choice, maybe the PEG as spec approach would be OK.

But, Proof is Possible

In fact, it **is** possible to connect the CFG and the PEG.

Results:

- **Soundness**. If the PEG parses a non-terminal N, creating a parse-tree along the way, then
  - the root of the tree is N
  - that tree is valid according to the CFG
  - its fringe is the input consumed by the PEG
- **Completeness**. If there is a valid parse-tree t with fringe f, then running the PEG on f will create t.

Corollary

As the PEG is deterministic, our CFG must be unambiguous.

Awkwardnesses

The proofs were entirely manual.

Changing the grammar to do more syntax (handling arrays, say), requires

- more specification and implementation work
- more manual proof

Conclusion

For a complete “end-to-end” story, a verified parser is a **very nice** thing to have.

For all its faults, the CFG remains the gold standard way to specify a language’s syntax.

PEGs are a reasonable way to implement parsers.

Handcrafted PEGs and CFGs can be connected by proof.