PEGs and Packrat Parsing

April 13, 2011
CS 132: Compiler Design

Chomsky Hierarchy

- ✓ Type 0: Unrestricted Grammars
- ✓ Type 1: Context-Sensitive Grammars
- ✓ Type 2: Context-Free Grammars
- ✓ Type 3: Regular Grammars

The formalism seems backwards if we want to recognize (parse) strings in the language…

Parsing Expression Grammars (PEGs)

Recognition-Based, rather than generative.

Define nonterminals as in CFGs, but right-hand-side can involve:

$e?$	Optional (greedy)
e^*	Zero-or-more (greedy)
e^+	One-or-more (greedy)
$\&e$	Matches (does not consume)
$!e$	Doesn't Match (does not consume)
$e_1\ e_2$	Sequencing
e_1 / e_2	Choice (prioritized)

Advantages:

- ✓ Deterministic by definition
- ✓ Linear-time parsing

Simple PEG Examples

$$A \leftarrow a\ b\ /\ a$$

$$B \leftarrow a\ /\ a\ b$$

$$C \leftarrow a\ a^*$$

$$D \leftarrow a^*\ a$$

$e?$	Optional (greedy)
e^*	Zero-or-more (greedy)
e^+	One-or-more (greedy)
$\&e$	Matches (does not consume)
$!e$	Doesn't Match (does not consume)
$e_1\ e_2$	Sequencing
e_1 / e_2	Choice (prioritized)

Restriction: Left Recursion

Normally, PEGs cannot be left-recursive:

E ← E + E / n

F ← n / E + E

(Recent work lifts this restriction, but we lose the linear-time-parsing guarantee.)

Unified Lexing and Parsing

PEGs commonly combine lexing and parsing in a single grammar.
(Regexp operators make this less painful.)

Advantages:

- ✓ Everything in one place with one formalism
- ✓ No problem with context-dependent lexing, as in
  - `vector<vector<int>> v(n>>2);`

Disadvantages:

- ✓ Need to specify whitespace in the PEG
  (E.g., after every lexical token)

```
Identifier ← Letter (Letter / Number)* Whitespace
Letter     ← [A-Za-z]
Number     ← [0-9]
Whitespace ← ([ \t\n] / Comment)*
```

Lexical Examples

Character literals

```
Literal ← ['] (![']  Char)* ['] 
Char    ← [\\] [nrt'"\[\]\\]
        / ![\\] .
```

Haskell comments

```
Comment ← "{-" (Comment  /  !"-}" .)* "-}"
```

End-of-file

```
EndOfFile ← !.
```

Parsing Examples (Whitespace omitted)

Expressions:

```
E ← T ( "+" E / "-" E)*
T ← F ( "*" F / "/" F)*
F ← Number / "(" E ")"
```

Unambiguous if-else:

```
Statement ← "if" "(" Expression ")" Statement "else" Statement
          / "if" "(" Expression ")" Statement
```

A Non-Context-Free PEG

$$A \leftarrow a\ A\ b\ /\ \varepsilon$$

$$C \leftarrow b\ C\ c\ /\ \varepsilon$$

$$S \leftarrow \&(A\ !b)\ a^{*}\ C\ !.$$

Packrat Parsing

Intuition: Follow the grammar (like recursive-descent), but:

1. Memoize: "Have I tried to match this nonterminal at this point in the string before? If so, how much of the input did it match?"

2. Backtracking is limited in PEGs;
   Each nonterminal can match a prefix of a string in exactly one way.

Packrat Parsing of AAABBBCCC

A ← a A b / ε
C ← b C c / ε
S ← &(A !b) a* C !.

	a	a	a	b	b	b	c	c	c	END
a	?5 1	?9 1	?13 1	?17 X						
b				?23 1	?27 1	?31 1	?36 X			
c							?60 1	?64 1	?68 1	
ε				?20 0			?57 0			
aAb	?4 6	?8 4	?12 2	?16 X						
A	?3 6	?7 4	?11 2	?15 0						
bCc				?49 6	?51 4	?53 2	?55 X			
C				?48 6	?50 4	?52 2	?54 0			
!b							?35 0			
&(A !b)	?2 0									
a*	?40 3	?41 2	?42 1	?43 0						
!.										?72 0
S	?1 9									

Application: Fortress Programming Language

- ✓ Designed for high-performance computing
- ✓ Developed by Guy Steele and others at Sun Labs (now Oracle)
- ✓ Relevance here: mathematical *and extensible* syntax.

```
cgit = 25
z := Vector[\E,n\](0)
r := x
rho := r DOT r
p := r
for j <- seq(1#cgit) do
 q = A p
 alpha = rho/(p DOT q)
 z += alpha p
 rho0 = rho
 r -= alpha q
 rho := r DOT r
 beta = rho / rho0
 p := r + beta p
end
(z, ||x - A z||)
```

$$
\begin{aligned}
&cgit = 25 \\
&z := \mathrm{Vector}[\![E, n]\!](0) \\
&r := x \\
&\rho := r \cdot r \\
&p := r \\
&\texttt{for } j \leftarrow seq(1 \,\#\, cgit) \texttt{ do} \\
&\quad q = A\,p \\
&\quad \alpha = \frac{\rho}{p \cdot q} \\
&\quad z \mathrel{+}= \alpha\,p \\
&\quad \rho_0 = \rho \\
&\quad r \mathrel{-}= \alpha\,q \\
&\quad \rho := r \cdot r \\
&\quad \beta = \rho/\rho_0 \\
&\quad p := r + \beta\,p \\
&\texttt{end} \\
&(z, \|x - A\,z\|)
\end{aligned}
$$

Extensible Syntax

> The Fortress programming language is intended to grow over time to accommodate the changing needs of its users [4]. One mechanism to allow for such growth is syntactic abstraction: It is possible to add new syntactic constructs to the language in libraries, defining new constructs in terms of old ones. In this manner, the language can gracefully adapt to unanticipated needs as they become apparent. Parsing of new constructs can be done alongside parsing of primitive constructs, allowing programmers to detect syntax errors in use sites of new constructs early. Programs in domain-specific languages can be embedded in Fortress programs and parsed along with their host programs. Moreover, the defini-

From Allen et al., "Growing a Syntax," 2009.

Extensible Syntax

> The parsing stage relies on *Parsing Expression Grammars* (PEGs) [16] because they have some advantages over usual Context-Free Grammars (CFG) [10] based parsing formalisms such as LL and LALR(k). PEGs are unambiguous, are closed under union, and integrates lexing with parsing. We need the closure property because we want to combine PEGs for different grammars and the integrated lexing and parsing makes it easy to achieve our goal of "similar syntax for definition and use". Furthermore, the parsers based on PEGs allow a linear execution time compared to the generalized CFG parsers. We briefly introduce PEGs in Section 5.1.1 along with a description of our pattern language which is effectively a variant of *Parsing Expressions*.

From Allen et al., "Growing a Syntax," 2009.

Extensible Syntax

> We have evaluated the design of our syntactic abstraction system by implementing the `for` loop example shown in Figure 3, and also a grammar that recognizes regular expressions and one recognizing XML. These examples show that the system is suitable for language extensions and domain specific languages by allowing their care-free implementation in general. The grammars for XML and regular expressions can be found as part of the open-source Fortress interpreter [3].
>
> The for-loop example demonstrates a weakness of the macro

From Allen et al., "Growing a Syntax," 2009.