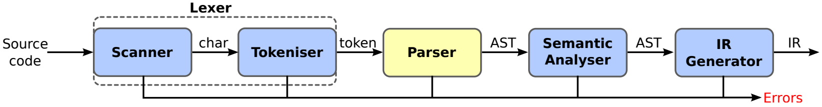
Compiling Techniques

Lecture 5: Top-Down Parsing

Christophe Dubach

6 October 2015

The Parser



- Checks the stream of words/tokens produced by the lexer for grammatical correctness
- Determine if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Used to build an IR representation of the code

Table of contents

Specifying syntax with a grammar

- Use Context-Free Grammar (CFG) to specify syntax

Contex-Free Grammar definition

A Context-Free Grammar G is a quadruple (S,N,T,P) where:

- S is a start symbol
- N is a set of non-terminal symbols
- T is a set of terminal symbols or words
- P is a set f production or rewrite rules where only a single non-terminal is allowed on the left-hand side (lhs)
  $P : N \rightarrow (N \cup T)^*$

From Regular Expression to Context-Free Grammar

- Kleene closure A^*:
  replace A^* to A_{rep} in all production rules and add
  $A_{rep} = A\ A_{rep} \mid \epsilon$ as a new production rule
- Positive closure A^+:
  replace A^+ to A_{rep} in all production rules and add
  $A_{rep} = A\ A_{rep} | A$ as a new production rule
- Option $[A]$:
  replace $[A]$ to A_{opt} in all production rules and add
  $A_{opt} = A \mid \epsilon$ as a new production rule

Example: function call

```
funcall ::= IDENT "(" [ IDENT ("," IDENT)* ] ")"
```

after removing the option:

```
funcall ::= IDENT "(" arglist ")"
arglist ::= IDENT ("," IDENT)*
          | ϵ
```

after removing the closure:

```
funcall ::= IDENT "(" arglist ")"
arglist ::= IDENT argrep
          | ϵ
argrep  ::= "," IDENT argrep
          | ϵ
```

To derive a syntactic analyser for a context free grammar express in an EBNF style:

- convert all the regular expressions as seen
- Implement a function for each non-terminal A.
  This function recognises sentences derived from A
- Recursion in the grammar corresponds to recursive calls of the created functions

This technique is called recursive-descent parsing or predictive parsing.

Parser class (pseudo-code)

```
Token token; // current token

void error(TokenClass... expected) {...}

boolean accept(TokenClass... expected) {
  return (token ∈ expected);
}

Token expect(TokenClass... expected) {
  Token result = token;
  if (accept(expected)) {
     nextToken();
     return result;
  }
  else
    error(expected);
```

CFG for function call

```
funcall ::= IDENT "(" arglist ")"
arglist ::= IDENT argrep
          | ϵ
argrep  ::= "," IDENT argrep
          | ϵ
```

Recursive-Descent Parser

```
void parseFunCall()
  expect(IDENT);
  expect(LPAR);
  parseArgList();
  expect(RPAR);

void parseArgList()
  if (accept(IDENT)) {
    nextToken();
    parseArgRep();
  }

void parseArgRep()
  if (accept(COMMA)) {
    nextToken();
    expect(IDENT);
    parseArgRep();
  }
```

Be aware of infinite recursion!

Left Recursion

```
E ::= E "+" T
    | T
```

The parser would recurse indefinitely!

Luckily, we can transform this grammar to:

```
E ::= T ("+" T)*
```

Consider the following bit of grammar

```
stmt    ::= assign ";"
          | funcall ";"
funcall ::= IDENT "(" arglist ")"
assign  ::= IDENT "=" lexp
```

```
void parseAssign() {
  expect(IDENT);
  expect(EQ);
  parseLexp();
}

void parseStmt() {
  ???
}
```

```
void parseFunCall() {
  expect(IDENT);
  expect(LPAR);
  parseArgList();
  expect(RPAR);
}
```

If it picks the wrong production, the parser may have to backtrack.
Alternative is to look ahead in input to pick correctly.

How much lookahead is needed?

- In general, an arbitrarily large amount

Fortunately:

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are LL(1) grammars.

LL(1)

Left-to-Right parsing;
Leftmost derivation;
1 symbol lookahead.

Basic idea: given $A \rightarrow \alpha|\beta$, the parser should be able to choose between α and β.

First sets

For some *rhs* $\alpha \in G$, define First(α) as the set of tokens that appear as the first symbol in some sting that derives from α:

$$x \in First(\alpha) \text{ iif } \alpha \rightarrow \cdots \rightarrow x\gamma, \text{ for some } \gamma$$

The *LL(1)* property: if $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like:

$$First(\alpha) \cap First(\beta) = \emptyset$$

This would allow the parser to make the correct choice with a lookahead of exactly one symbol! (almost, see next slide!)

What about ϵ-productions (the ones that consume no token)?

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\epsilon \in \mathit{First}(\alpha)$, then we need to ensure that $\mathit{First}(\beta)$ is disjoint from $\mathit{Follow}(\alpha)$.

$\mathit{Follow}(\alpha)$ is the set of all words in the grammar that can legally appear immediately after an α (see EaC§3.3 for details on how to build the *First* and *Follow* sets).

Let's define $\mathit{First}^+(\alpha)$ as:

- $\mathit{First}(\alpha) \cup \mathit{Follow}(\alpha)$, if $\epsilon \in \mathit{First}(\alpha)$
- $\mathit{First}(\alpha)$ otherwise

LL(1) grammar

A grammar is *LL(1)* iff $A \rightarrow \alpha$ and $B \rightarrow \beta$ implies:

$$\mathit{First}^+(\alpha) \cap \mathit{First}^+(\beta) = \emptyset$$

Given a grammar that has the *LL(1)* property:

- can write a simple routine to recognise each *lhs*
- code is both simple and fast

Predictive Parsing

Grammar with the *LL(1)* property are called *predictive grammars* because the parser can "predict" the correct expansion at each point. Parsers that capitalise on the *LL(1)* property are called *predictive parsers*. One kind of predictive parser is the *recursive descent* parser.

Sometimes, we might need to lookahead one or more tokens.

LL(2) Grammar Example

```
stmt    ::= assign ";"
          | funcall ";"
funcall ::= IDENT "(" arglist ")"
assign  ::= IDENT "=" lexp
```

```
void parseStmt() {
  if (accept(IDENT)) {
    if (lookAhead(1) == LPAR)
      parseFunCall();
    else if (lookAhead(1) == EQ)
      parseAssign();
    else
      error();
  }
  else
    error();
}
```

Next lecture

- More about LL(1) & LL(k) languages and grammars
- Dealing with ambiguity
- Left-factoring
- Bottom-up parsing