Compiler Construction

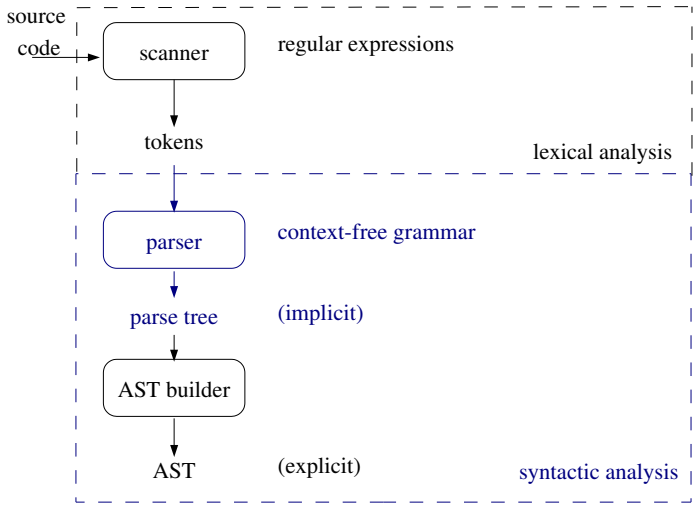
Top-down Parsing

Lennart Andersson

Revision 2012–01–25

2012

CFGs



A typical language definition

Syntax

- EBNF and RE

Semantics

- Described using natural language, referring to the syntax
- Static semantics: meaning and rules that can be computed at compile-time
- Dynamic semantics: meaning and rules that can be computed at runtime

Example: The Java language specification

- http://java.sun.com/docs/books/jls/

Recursive descent parsing

Assume that grammar has exactly one production for each non-terminal symbol and that any choice between possible alternatives can be made by looking on the next k tokens. k should small; say $k \leq 2$.

- Define one parsing method for each nonterminal symbol.
- Use the name of the symbol as method name.
- The body of the method should accept all right hand parts for productions for this nonterminal using parsing methods for all nonterminal symbols.
- If a method fails to accept the input it should report an error.

Grammar fragment

statement → assignment | compundStmt
assignment → ID '=' expr ';'
compoundStmt → '{' statement* '}'

Auxiliary declarations

```
class Parser {
  final static int ID=1, WHILE=2, DO=3, ASSIGN=4, ...;
  private int token; // current token
  private int nextToken() {...};
  void error(String string) {...};
  void accept(int t) {
    if (token==t) {
      token = nextToken();
    } else {
      error("expected: " + t + "found: " + token);
    }
  }
  ...
```


Parsing method for assignment

assignment → ID '=' expr ';'

- Assume that there is a parsing method `expr()` that accepts an expression or reports en error.
- Assume that `token` contains the first token that haven't been accepted yet.

```
void assignment() {
  accept(ID);
  accept(ASSIGN);
  expr();
  accept(SEMICOLON);
}
```


Parsing method for compoundStmt

compoundStmt → '{' statement* '}'

```
void compoundStmt() {
  accept(LEFTBRACE);
  while(token!=RIGHTBRACE) {
    statement();
  }
  accept(RIGHTBRACE);
}
```

Parsing method for statement

statement → assignment | compoundStmt

```
void statement() {
  switch(token) {
  case ID: assignment();
    break;
  case LEFTBRACE: compoundStmt();
    break;
  default: error("expecting statement found " + token);
  }
}
```


Common prefix – 1

statement → ID '=' expr ';'
statement → ID '(' expr ')' ';'

Use more lookahead.

```
void statement() {
  switch(token) {
  case ID:
      if(lookahead(2)==ASSIGN) {
         assignment();
      } else {
         callStmt();
      }
    break;
  case LEFTBRACE: compoundStmt();
```

Requires a pushback scanner. Might be somewhat less efficient.

Common prefix – 2

statement → ID '=' expr
statement → ID '(' expr ')'

Rewrite the grammar.

statement → ID ('=' expr | '(' expr ')')

Slightly less readable.

Left recursion – 1

expr → expr '-' ID | ID

What's the problem?

There is a common prefix, but lookahead=2 suffices, but:

```
void expr() {
  if(lookahead(2)==MINUS) {
    expr() ; accept(MINUS); accept(ID);
  } else {
    accept(ID);
  }
}
```

will loop forever on `ID - ID ;`

Left recursion – 2

expr → expr '-' ID | ID

Rewriting

expr → ID '-' expr | ID

What's the problem?

The parser will not respect the associativity for '-'. The AST builder has to fix this.

Left recursion – 3

expr → expr '-' ID | ID

Use iteration

expr → ID ('-' ID)*

```
void expr() {
   accept(ID);
   while (token==MINUS) {
     accept(MINUS); accept(ID);
   }
}
```

Readable. Not so hard to build the AST.

A CFG for simple arithmetic expressions

expr → expr '*' expr
expr → expr '+' expr
expr → '(' expr ')'
expr → ID
expr → INT

What's the problem?

Ambiguous, left recursive, common prefixes.

Rewriting expression grammars

Restrict certain subtrees by introducing new nonterminals for them.

Precedence

- Introduce a new nonterminal for each precedence level. Start with the operator with lowest precedence.

Associativity

- Left associativity: Restrict the right operand so it can only contain subtrees of a higher precedence.

Standard rewriting for expressions

expr → term ('+' term)*
term → factor ('*' factor)*
factor → ID | INT | '(' expr ')'

Readable
Not so hard to build AST

JavaCC specifications

statement → assignment | compoundStmt
assignment → ID '=' expr ';'
compoundStmt → '{' statement* '}'

```
void statement() : {}
{
  assignment() | compoundStmt()
}
void assignment() : {}
{
  <ID> <ASSIGN> expr() <SEMICOLON>
}
void compoundStmt() : {}
{
  <LEFTBRACE> (statement())* <RIGHTBRACE>
}
```


JavaCC specifications

When doing semantic analysis it is better to extract <ID> using a method to avoid code duplication.

```
void assignment() : {}
{
  id() <ASSIGN> expr()
}
void id() : {}
{
  <ID>
}
```


Lookahead

We may use lookahead to discriminate between an assignment and a procedure call in

statement → assignment | callStmt | whileStmt
assignment → ID '=' expr';'
callStmt → ID '(' expr ')' ';'

```
void statement() : {}
{
LOOKAHEAD(2)  assignment()
| callStmt()
| whileStmt()
}
```

JavaCC will use the extra lookahead before selecting assignment(). If that fails then a single token lookahead will be used in the following alternatives.

Expressions

expr → term ('+' term)*
term → factor ('*' factor)*
factor → ID | INT | '(' expr ')'

```
void expr() : {}
{
  term() (<PLUS> term())*
}
void term() : {}
{
  factor() (<TIMES> factor())*
}
void factor() : {}
{
  id() | intExpr() | <LEFTPAR> expr() <RIGHTPAR>
}
```


Algorithm for parser construction

What's the problem?

- Detecting choice points and conflicts.
- How much lookahead is needed?
- Is the grammar ambiguous?

Choices

Assume that we have a grammar on canonical form.

statement → assignment
statement → compoundStmt
assignment → ID '=' expr ';'
compoundStmt → '{' statements '}'
statements → statement statements
statements → ϵ

It will be easy to decide which production to use if it is enough to consider the next token only.
Let $FIRST(\gamma)$ be the set of terminal symbols that can be the first symbol in a string generated by γ. So

$$FIRST(assignment) = \{ID\}$$
$$FIRST(compoundStmt) = \{'\{'\}$$

The choice for *statement* is easy since these sets are disjoint.

Choices

How about

statements → statement statements
statements → ϵ

$$FIRST(statement\ statements) = \{ID, '\{'\}$$
$$FIRST(\epsilon) = \emptyset$$

Is there a problem?
Well, if the next token after the string generated by statements could be one of $\{ID, '\{'\}$ there is.
We need to define $FOLLOW(X)$ to be the set of terminal symbols that can follow a string generated by X.

$$FOLLOW(statements) = \{'\}'\}$$

Choices

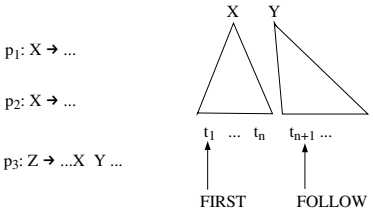
When do we need to know FOLLOW(X)?

When X may generate the empty string.

Define

$$NULLABLE(X) = \begin{cases} true, & \text{if } X \Rightarrow^* \epsilon \\ false, & \text{otherwise} \end{cases}$$

How to select the production to use



Which tokens can occur in the FIRST position?
Can X derive the empty string? NULLABLE(X)?
If so, which tokens can occur in the FOLLOW position?

"Definition" of $FIRST(\gamma)$

Let $t \in T$, $X \in N$, $s \in N \cup T$, and $\gamma \in (N \cup T)^*$.

$$\begin{aligned} FIRST(\epsilon) &= \emptyset \\ FIRST(t) &= \{t\} \\ FIRST(s\,\gamma) &= FIRST(s), \text{ if } NULLABLE(s) = false \\ FIRST(s\,\gamma) &= FIRST(s) \cup FIRST(\gamma), \text{if } NULLABLE(s) = true \\ FIRST(X) &= \text{the union of all } FIRST(\gamma) \text{ where} \\ &\quad X \to \gamma \text{ is a production} \end{aligned}$$

Why "Definition"? It is recursive and the recursion might not terminate.

Dangling else, again

Canonical grammar without common prefixes

$$\begin{aligned} p_0\text{: ifStmt} &\to \text{'if' expr 'then' statement optElse} \\ p_1\text{: optElse} &\to \epsilon \\ p_2\text{: optElse} &\to \text{'else' statement} \end{aligned}$$

$$\begin{aligned} FIRST(optElse) &= \{\texttt{else}\} \\ FOLLOW(optElse) &= FOLLOW(statement) = \{\texttt{else}, \texttt{;}, \ldots\} \end{aligned}$$

There is a conflict; we cannot choose between p_1 and p_2 by just looking at the next token when it is ELSE. Since the grammar is ambigous we cannot decide without asking the language designer.

Example

p_0 : *start* → *program* \$
p_1 : *program* → 'begin' *body* 'end'
p_2 : *body* → *optStatements*
p_3 : *optStatements* → *statement moreStatements*
p_4 : *optStatements* → ϵ
p_5 : *moreStatements* → ';' *statement moreStatements*
p_6 : *moreStatements* → ϵ
p_7 : *statement* → 'if' *expr* 'then' *statement*
p_8 : *statement* → *ID* '=' *expr*
p_9 : *expr* → *ID*
p_{10} : *expr* → *INT*

\$ denotes the token returned by the scanner at end of file.

NULLABLE, FIRST, and FOLLOW

	NLBL	*FIRST*	*FOLLOW*
program			
body			
optStatements			
moreStatements			
statement			
expr			

p_1 : *program* → 'begin' *body* 'end'
p_2 : *body* → *optStatements*
p_3 : *optStatements* → *statement moreStatements*
p_4 : *optStatements* → ϵ
p_5 : *moreStatements* → ';' *statement moreStatements*
p_6 : *moreStatements* → ϵ
p_7 : *statement* → 'if' *expr* 'then' *statement*
p_8 : *statement* → *ID* '=' *expr*
p_9 : *expr* → *ID*
p_{10} : *expr* → *INT*

LL(1) table

	begin	*end*	;	*if*	*then*	*ID*	=	*INT*	\$
start									
program									
body									
optStatements									
moreStatements									
statement									
expr									

Construction of LL(1) table

Initialize all elements in the table to the empty set.

```
for each production p : X → γ
  for each t ∈ FIRST(γ)
    add p to table[X, t]
  if NULLABLE(γ)
    for each s ∈ FOLLOW(X)
      add  p to table[X, s]
```

If some entry has more than one element then the grammar is not LL(1).

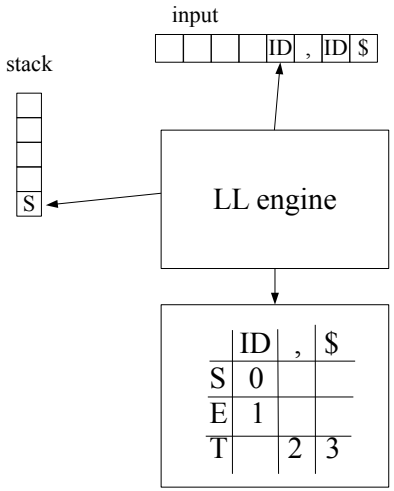
LL(1) parsing engine (almost Java)

```
stack.push($); stack.push(start);
token = next();
repeat {
  top = stack.top()
  if (top.isTerminal()) {
    if(top.equals(token)) {
      stack.pop(); token = next();
    } else error();
  } else { // top is nonTerminal
    if (table[top, token] == top -> Y1 Y2 ... Yk) {
      stack.pop();
      stack.push(Yk); ... stack.push(Y1);
    } else error();
  }
} until (top.equals($))
```


Explanation

- stack can hold terminals and nonterminals
- the end of file marker is terminal
- next() returns the next token from the scanner
- table is the parsing table. It is indexed by one nonterminal and one terminal symbol.
- each table entry contains a production or is empty. The left hand part of the production equals the row index and is redundant. The right hand part is a list of terminals and nonterminals. The pseudo code uses pattern matching inspired by Haskell.
- if the entry is nonempty the symbols are stacked in reverse order
- if the entry is empty then an error is reported

LL(1) parsing engine



	ID	,	$
S	0		
E	1		
T		2	3

0: S → E $
1: E → ID T
2: T → , ID T
3: T → ε

Left recursive grammar

p_1 : *term* → *term*'*'*factor*
p_2 : *term* → *factor*
p_3 : *factor* → *ID*
p_4 : *factor* → *INT*

	NLBL	*FIRST*
term	*F*	*ID INT*
factor	*F*	*ID INT*

	*	*ID*	*INT*
term		p_1 p_2	p_1 p_2
factor		p_3	p_4

A left recursive grammar is not LL(k).

Choice points in EBNF grammar

- multiple productions for some nonterminal
- alternatives |
- optionals [ ] or ?
- iteration $(\ldots)^*$ or $(\ldots)^+$

The recursive descent parser must be able to make the correct choice looking at the next token (or the next k tokens).

If JavaCC suggests using more lookahead then compute FIRST and FOLLOW sets at the choice point before following the advice.

Grammar with common prefix

p_1 : *expr* → *factor*'*'*expr*
p_2 : *expr* → *factor*
p_3 : *factor* → *ID*
p_4 : *factor* → *INT*
p_5 : *factor* → "("*expr*')'

	NLBL	*FIRST*
expr	*F*	*ID INT (*
term	*F*	*ID INT (*
factor	*F*	*ID INT (*

	*	ID	INT	(	)
expr		p_1, p_2	p_1, p_2	p_1, p_2	
factor		p_3	p_4	p_5	

A grammar with common prefixes is not LL(1).
It may be LL(k). This one is not. Why?

Summary questions

- Construct a CFG for a part of a programming language that you are familiar with.
- Construct recursive descent parsers for conventional programming language constructs.
- Give typical examples of ambiguities in CFGs
- What is the difference between LL(1) and LL(k)?
- What is a "common prefix" and how can it be eliminated?
- What is "left recursion" and how can it be eliminated?
- In what way can an LL syntax tree differ from the desired AST?
- Construct a EBNF grammar for conventional arithmetic expressions that respect standard precedence and associativity.
- What is NULLABLE(X), FIRST(X), and FOLLOW(X)?
- Construct an LL(1) table from NULLABLE, FIRST, and FOLLOW tables.
- True or false? If some element in the LL(1) table has more than one element then the grammar is ambiguous.

Readings

F4: Parsing. More about context-free grammars.
- Appel, 3-3.1

S2: Seminar 2. Grammars and parsing.
- Study F3, F4 and associated readings.

L2: Lab 2. Parsing
- Study the lab material, prepare carefully

F5: LL() and AST
- Appel,