Compiler Construction

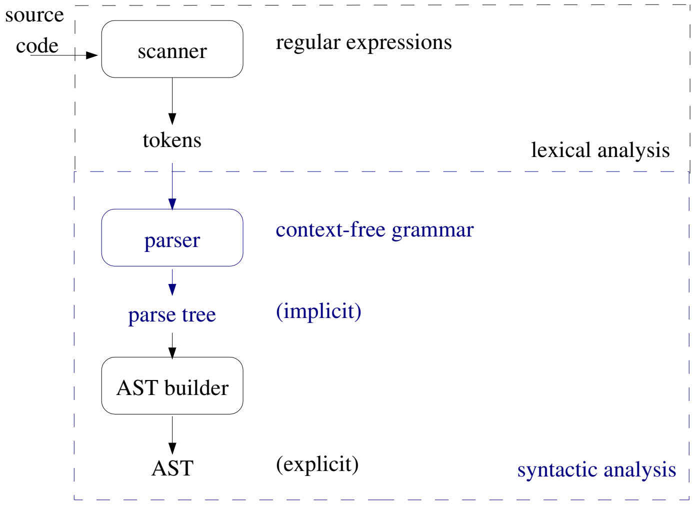
Grammars

Lennart Andersson

Revision 2012–01–23

2012

Parsing



source code

scanner

regular expressions

tokens

lexical analysis

parser

context-free grammar

parse tree

(implicit)

AST builder

AST

(explicit)

syntactic analysis

Regular expressions vs Grammars

Regular expressions are not powerful enough to describe the syntax of programming languages.

A simple generalization can be made: Use names for regular expressions and allow theses names to appear in expressions (recursively).

expr $\rightarrow$ INT | expr '+' expr | '(' expr ')'

Rules of the grammar G_{Stmt}

"while (k<=n) {sum = sum+k; k=k+1;}"

statement $\rightarrow$ whileStmt | assignment | compoundStmt
whileStmt $\rightarrow$ 'while' . . .
assignment $\rightarrow$
compoundStmt $\rightarrow$
expr $\rightarrow$
lessEqual $\rightarrow$
add $\rightarrow$

Grammar

A *context-free grammar* has four components, $G = (N, T, P, S)$

- N – finite set of nonterminal symbols
- T – finite set of terminal symbols (tokens)
- P – finite set of productions (rules)
- S – the start symbol (one of the nonterminals in N).

N and T are disjoint sets.

Each production has a left part that is a nonterminal symbol and a right part that is a regular expression with terminal and nonterminal symbols.

There are three operations: alternative (|), concatenation, and iteration (*) in order of increasing precedence.

Parentheses are used for grouping.

Derivation

Derivation of "while (k<=n) {sum = sum+k; k=k+1;}"

statement ⇒ whileStmt ⇒
'while' '(' expr ')' statement ⇒
'while' '(' lessEqual ')' statement ⇒
'while' '(' expr '<=' expr ')' statement ⇒
'while' '(' ID '<=' expr ')' statement ⇒
'while' '(' ID '<=' ID ')' statement ⇒
'while' '(' ID '<=' ID ')' compoundStmt ⇒
. . .

All the strings in the derivation are called *sentential forms*.
The final string is called a *sentence*.

Parse tree

statement

while (ID <= ID) { ID = ID + ID ; ID = ID + INT ; }

Derivation of strings

A string belongs to a language $\mathcal{L}(G)$ if it can be derived from the start symbol. Derivation (härledning):

- Start with the start symbol
- Replace a nonterminal X using the right-hand side of a production for X:
  - if there are alternatives (|), choose one
  - if there are iterations (*), repeat any number of times
- Continue replacing nonterminals in this way until there are only terminal symbols left
- The resulting string belongs to the language

Derivations

A derivation has the form

$$\gamma_0 \Rightarrow \gamma_1 \cdots \Rightarrow \gamma_n \ , \qquad n \geq 0$$

where each γ_i is a string of terminal and non-terminal symbols obtained from the previous one by the use of one production.

When there is such a derivation we write

$$\gamma_0 \Rightarrow^* \gamma_n$$

The language $\mathcal{L}(G)$

The language $\mathcal{L}(G)$, where $G = (N, T, P, S)$, is the set of all strings that can be derived from S using the rules in P. G *generates* $\mathcal{L}(G)$.

$$\mathcal{L}(G) = \{ w \in T^* \mid S \Rightarrow^* w \}$$

T^* denotes the set of all strings with symbols from T.
G is finite but $\mathcal{L}(G)$ is usually an infinite set.

Leftmost derivation (vänsterhärledning)

Grammar G_{expr}

$$\begin{array}{lcl} \text{expr} & \rightarrow & \text{expr '+' expr} \\ \text{expr} & \rightarrow & \text{expr '*' expr} \\ \text{expr} & \rightarrow & \text{INT} \end{array}$$

The leftmost nonterminal is replaced in each derivation step.

Derive INT + INT * INT.

expr $\Rightarrow$
expr '+' expr $\Rightarrow$
INT '+' expr $\Rightarrow$
INT '+' expr '*' expr $\Rightarrow$
INT '+' INT '*' expr $\Rightarrow$
INT '+' INT '*' INT

Another leftmost derivation

Grammar G_{expr}

$$\begin{array}{lcl} \text{expr} & \rightarrow & \text{expr '+' expr} \\ \text{expr} & \rightarrow & \text{expr '*' expr} \\ \text{expr} & \rightarrow & \text{INT} \end{array}$$

Derive INT + INT * INT.

Parse tree

expr $\Rightarrow$
expr '*' expr $\Rightarrow$
expr '+' expr '*' expr $\Rightarrow$
INT '+' expr '*' expr $\Rightarrow$
INT '+' INT '*' expr $\Rightarrow$
INT '+' INT '*' INT

Ambiguous grammar

A grammar is *ambiguous* if there is a string having more than one parse tree.

A grammar is *unambiguous* if there is no string having more than one parse tree.

If a grammar is ambiguous we should try to find a grammar generating the same language that is unambiguous. In the current case we would prefer a parse tree that respects operator precedences.

Equivalent grammars

Two grammars, G_1 and G_2, are equivalent if they generate the same language, i.e. each sentence that can be derived using one of the grammars, can also be derived using the other grammar.

$$\mathcal{L}(G_1) = \mathcal{L}(G_2)$$

An equivalent grammar

expr → expr '+' term
expr → term
term → term '*' factor
term → factor
factor → INT

Derive INT + INT * INT.

expr ⇒ expr '+' term ⇒
term '+' term ⇒ factor '+' term ⇒
INT '+' term ⇒ INT '+' term '*' factor ⇒
INT '+' factor '*' factor ⇒ INT '+' INT '*' factor ⇒
INT '+' INT '*' INT

Other equivalent grammars

expr → expr '+' term | term
term → term '*' factor | factor
factor → INT

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

Different notations for CFGs

Canonical form (kanonisk form)

- The simplest notation: no alternatives nor iterations are permitted.
- Useful when proving formal properties of grammars and implementing parser generators.

BNF – Backus-Naur form

- Includes a shorthand for writing several productions for the same nonterminal in the same rule

EBNF – Extended Backus-Naur Form

- Additional shorthand notations are introduced: regular expressions can be written in the right-hand side of a rule, e.g., (...)*
- gives very compact grammars
- standard notation for describing the syntax for programming languages
- used in JavaCC

Transformation to canonical form – iteration

A grammar rule containing iteration on the top level has the form

$$\mathrm{X} \rightarrow \gamma_1\ \gamma_2{}^{*}\ \gamma_3$$

where γ_i is a regular expression on the alphabet $N \cup T$.

It can be transformed into

$$\begin{array}{lll} \mathrm{X} & \rightarrow & \gamma_1\ \mathrm{N}\ \gamma_3 \\ \mathrm{N} & \rightarrow & \gamma_2\ \mathrm{N} \\ \mathrm{N} & \rightarrow & \epsilon \end{array}$$

where N is a new nonterminal symbol.

Transformation to canonical form – alternatives

A grammar rule containing an alternative on the top level has the form

$$\mathrm{X} \rightarrow \gamma_1 \mid \gamma_2$$

can be transformed into

$$\begin{array}{lll} \mathrm{X} & \rightarrow & \gamma_1 \\ \mathrm{X} & \rightarrow & \gamma_2 \end{array}$$

Example

expr → term (('+' | '-') term)* can be transformed into

EBNF

In EBNF (JavaCC) other operators are permitted
- γ^+ is equivalent to $\gamma\ \gamma^*$
- $\gamma?$ and $[\gamma]$ are equivalent to $\gamma \mid \epsilon$

Example, dangling else

statement → 'if' expr 'then' statement ['else' statement]

This production introduces ambiguity. Consider

if e1 then if e2 then s1 else s2

if e1 then {if e2 then s1} else s2
if e1 then {if e2 then s1 else s2}

Dangling else – remedies

statement → 'if' expr 'then' statement ['else' statement] 'fi'

statement → matched | unmatched
matched → 'if' expr 'then' matched 'else' statement
matched → otherStmts
unmatched → 'if' expr 'then' statement
unmatched → 'if' expr 'then' matched 'else' unmatched

Rightmost derivation

Grammar G_{expr}

expr → expr '+' expr
expr → expr '*' expr
expr → INT

The rightmost nonterminal is replaced in each derivation step.

Derive INT + INT * INT.

expr ⇒ expr '+' expr
expr '+' expr '*' expr ⇒
expr '+' expr '*' INT ⇒
expr '+' INT '*' INT ⇒
INT '+' INT '*' INT

LL and LR parsers

Parser algorithms will be the topic of the next lecture.

- an LL parser constructs a leftmost derivation
- an LR parser constructs a rightmost derivation

Is $\mathcal{L}(G_{Stmt})$ too large?

statement	$\rightarrow$	whileStmt | assignment | compoundStmt
whileStmt	$\rightarrow$	'while' '(' expr ')' statement
assignment	$\rightarrow$	ID '=' expr ';'
compoundStmt	$\rightarrow$	'{' statement* '}'
expr	$\rightarrow$	lessEqual | add | ID | INT
lessEqual	$\rightarrow$	expr '<=' expr
add	$\rightarrow$	expr '+' expr

G_{Stmt} allows statements that would not be legal in Java.

```
while (a+b) { ...}
x = x <= y;
```

How can this be handled?

What is meant by context-free?

- A canonical grammar is *context-free* if every production has the form $A \rightarrow \gamma$, where A is a nonterminal and γ is a string of terminals and nonterminals.
- A production on the form $\alpha A \beta \rightarrow \alpha \gamma \beta$, where α and β are strings of terminals and nonterminals, is *context-sensitive*. A may be replaced by γ in the context $\alpha \cdot \beta$.

Comparing CFGs to REs

	CFG	RE
Alphabet	terminal symbols (tokens)	characters
Language	sentences (sequences of tokens)	strings (sequences of characters)
Used for describing	syntax for programming languages	tokens
Expressive power	recursion	iteration
Recognizer	nondeterministic pushdown automaton (NFA with stack)	deterministic finite automata (DFA)

The Chomsky hierarchy

Grammar	Rule patterns	Type
regular	$X \rightarrow aY$ or $X \rightarrow \epsilon$	3
context free	$X \rightarrow \gamma$	2
context sensitive	$\alpha\, X\, \beta \rightarrow \alpha\, \gamma\, \beta$	1
arbitrary	$\gamma \rightarrow \delta$	0

Regular grammars have the same describing power as regular expressions.
Type 2 and 3 are of practical use in compiler construction. The others are only of theoretical interest.

Parts of a typical CFG for Java

CompilationUnit → [*PackageDeclaration*](*ImportDeclaration*)*
(*TypeDeclaration*)*
PackageDeclaration → 'package' *Name* ';'
ImportDeclaration → 'import' *Name* ["." '*'] ';'
TypeDeclaration → *ClassDeclaration*
| *InterfaceDeclaration* | ';'
ClassDeclaration → ("abstract" | 'final'
| "public")* *CleanClassDeclaration*
CleanClassDeclaration → 'class' *ID*["extends" *Name*]
["implements" *NameList*] *ClassBody*
ClassBody → '{'(*ClassBodyDeclaration*)*'}'
...
Name → *ID*("." *ID*)*
NameList → *Name*("," *Name*)*
...

Unsolvable problems

- Given two context-free grammars, G_1 and G_2. Is $\mathcal{L}(()G_1) = \mathcal{L}(()G_2)$?
- Given a context-free grammar, G. Is G ambiguous?

Summary questions

- Define a small example language with a context-free grammar.
- What is a nonterminal symbol? A terminal symbol? A production? A start symbol?
- Given a grammar G, what is meant by the language $\mathcal{L}(G)$?
- What is a derivation? A leftmost derivation?
- When are two grammars equivalent?
- What is the difference between a context-free grammar and a context-sensitive grammar?
- When should we use canonical form, and when EBNF?
- Translate an EBNF grammar to canonical form.
- Explain why context-free grammars are more powerful than regular expressions.
- Explain why tokens are usually defined by regular expressions rather than context-free grammars.

Some terms

abbr.	English	Svenska
AST	abstract syntax tree	abstrakt syntaxträd
	terminal symbol	terminalsymbol
	nonterminal symbol	icketerminalsymbol
CFG	context-free grammar	kontextfri grammatik
	context-sensitive grammar	kontextberoende grammatik
	derivation	härledning
	canonical form	kanonisk form
BNF	Backus-Naur form	
EBNF	Extended BNF	utvidgad BNF
RE	regular expression	reguljärt uttryck
ϵ	the empty string	tomma strängen
DFA	deterministic finite automaton	deterministisk ändlig automat

Readings

- F3: Context free grammars, derivations, ambiguity, EBNF
  Appel, chapter 3-3.1.
- F4: Predictive parsing. Recursive descent. LL grammars and parsing Left recursion and factorization.
  Appel, chapter 3.2
- Seminar 2. Grammars.
- Programming assignment 1. Build a scanner. Instructions is on the web.