Lecture 2-3: Syntax definition, syntax analysis

- Formal syntax definitions
- Lexical and context free syntax
- Ambiguity in context free syntax
- Syntax graphs
- Goals for syntax analysis
- Basic ideas, complexity of SA
- Syntax analysis in pictures
- Top-down syntax analysis, Lookahead
- Bottom-up syntax analysis

Formal syntax definitions

The syntax of a programming language defines the set of all strings that can be seen as programs (meaning not considered).

- There is a distinction between syntax (form) and semantics (meaning)
- Syntactical correctness is necessary but not sufficient for semantical correctness
- Defining syntax is considerably easier than defining semantics

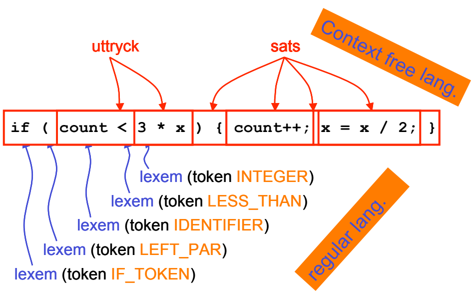
Formal syntax definitions

- A programming language is a formal language
  ⇒ the syntax can be defined formally (=unambiguous) e.g. using grammars
- Advantages with a formal syntax definition
  - exact
  - ”standard” for computer scientists, everybody knows how to read it
  - easier to produce compilers that are compatible w e o
  - automatic generation of (the front end of) compilers
  - No doubt about the (syntactical) correctness of programs
  - Algorithms and results from formal language theory can be used
  - Makes it easier to define the semantics

Lexical and context free syntax

- Syntax definitions are often divided into two levels:
  - lexical syntax
  - context free syntax
- The lexical level consists of simple syntactic categories for identifiers, numbers, keywords, …
  Can be defined by regular expressions
  Central concepts:
  - token = one of the categories (e.g. integer)
  - lexeme = a string in one of the categories (e.g. 5317)
- Syntactic categories of higher complexity is treated on the context free level (statements, expressions, …). Often defined by a grammar

Example



Ambiguity

The context free syntax is ambiguous if there exists a program with more than one derivation/ parse tree (= more than one leftmost derivation)

- Leftmost derivation = a derivation where you replace the leftmost nonterminal in the string in each step
- An ambiguous grammar is not erroneous but causes problems when semantics should be defined
- Example: <expr> → <expr> * <expr>
  | <expr> + <expr>
  | <identifier> | <integer>

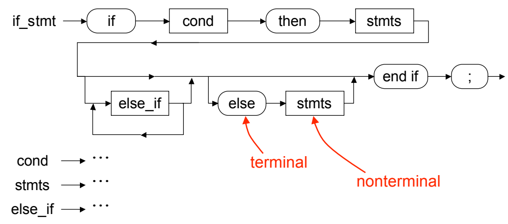
Ambiguity (2)

Some common techniques to avoid ambiguity (which can be built into the grammar)

- Binding rules (operator precedence)
  e.g. ^ binds stronger than * which binds stronger than +.
- Right or left associative operators
  e.g. a + b + c means (a + b) + c since + is left associative
- Parentheses
  e.g. (a + b) * (a + c) to overrule the binding rules

Syntax graphs (an example)

Syntax graphs are equivalent to context free grammars.
Ada's if-then-else:



Extended BNF

Some common techniques to get more compact grammars

- Optional parts is marked by [ ]
  IF -> if E then S [else S]
- Repetition (zero or more times)
  IDLIST -> ID {, ID}
- Choice (one of)
  T -> T (* | / | %) F

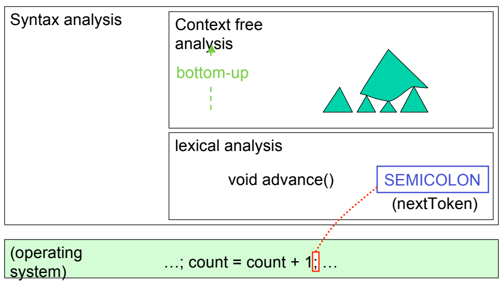
Syntax analysis - goals

- Check syntactic correctness
  - The most fundamental task of the parser.
- Build a derivation tree (explicit or implicit)
  - The tree is later used to translate / interpret the program
- If possible, continue the analysis when errors are detected (recovery)
  - A program with 386 errors should not need 387 tries to pass compilation
- Give useful error messages
  - Where does the error appear?
  - What was expected?
  - Suggest how to handle it (in case of a "standard error")

The parsing process and its complexity

- The lexical/context free levels are also found in the structure of the parser
  - lexical analysis = routines which reads the input string and produce tokens
    = a finite automaton (in principle)
  - C.F. analysis = routines which read tokens and build a derivation tree
- Context free analys is done top-down or bottom-up
  - top-down starts in the root of the derivation tree (works forwards)
  - bottom-up starts in the leaves of the tree (works backwards)
- Complexity is important
  - generally $O(n^3)$ for context free languages (too high!)
  - $O(n)$ for LL- and LR-grammars

Syntax analysis in pictures



Recursive descent parsing (example)

S → P { (+ | –) P }
P → F { (* | /) F }
F → <number> | (S)

EBNF (extended Backus-Naur form)
{…} = repeat 0,1,2,… times
…|… = choice between alternatives

Every nonterminal becomes a subprogram:

```
void F() {
  if (lex.nextToken==NUMBER) lex.advance();
  else if (lex.nextToken==LEFT_PAR) {
    lex.advance(); S();
    if (lex.nextToken==RIGHT_PAR) lex.advance();
    else error("missing ')'");
  }
  else error("number or '(' expected");
}
```


Top-down: Pros and cons

Advantages
- easy to implement (also by hand)
- fast (i.e. linear time)
- Most times the class of LL-grammars is big enough

Disadvantages
- The grammar must not be left recursive
- You need "lookahead k" (see next slide); but normally k=1 is sufficient
- The class of LR-grammars (basis for bottom-up analysis) can describe more languages than LL-grammars

Lookahead

STMT → (CALL | ASSIGN) { ; STMT }
CALL → <IDENTIFIER> (PARAM)
ASSIGN → <IDENTIFIER> = EXPR
PARAM → …
EXPR → …

```
void STMT() {
  if (???) CALL(); ELSE ASSIGN();
  while (lex.advance==SEMICOLON) {
    lex.advance(); STMT();
  }
}
```

In `(???)` we must check the token after `nextToken` ,´(´ or ´=´

⇒ We need lookahead 2, the grammar is an LL(2)-grammar

An LL-grammar means an LL(1)-grammar

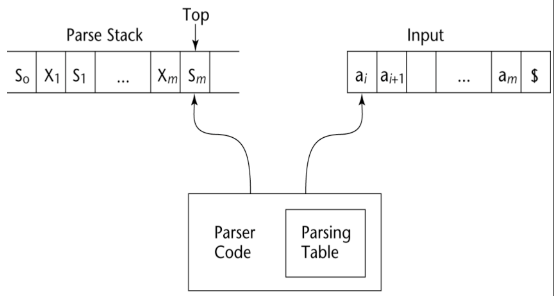
Top-down parsing, cont

- FIRST(A) = all terminal symbols that can begin strings that are derived from A
- Requirements for top-down parsing
  1) Not left recursive
  2) Disjoint FIRST sets for possible choices of RHS
- Solutions
  1) Recursive descent: use EBNF
     Table driven: rewrite the grammar
  2) Left factoring
- Table driven top-down parsing (LL(1))
  - A stack with nonterminals to be expanded
  - A table w `nextToken` / stack symbol which controls parsing. For every symbol in FIRST(A) there's an entry in the table

Bottom-up syntax analysis

- The input string is reduced to the start symbol by using the productions from right to left (producing the reverse of a derivation)
- Symbols still not reduced (plus some status information) are stored on a stack
- Two kinds of parsing steps: ´shift´ and ´reduce´
- The topmost symbols in the stack and `nextToken` decides which parsing action to do (→ LR(1))
- Information about what to do is found in the tables ´action´ och ´goto´

Structure of An LR Parser



Bottom-up Parsing

- Initial configuration: $(S_0, a_1 \ldots a_n\$)$
- Parser actions:
  - If ACTION[S_m, a_i] = Shift S, the next configuration is: $(S_0X_1S_1X_2S_2 \ldots X_mS_ma_iS, a_{i+1} \ldots a_n\$)$
  - If ACTION[S_m, a_i] = Reduce $A \rightarrow \beta$ and S = GOTO[S_{m-r}, A], where r = the length of β, the next configuration is $(S_0X_1S_1X_2S_2 \ldots X_{m-r}S_{m-r}AS, a_ia_{i+1} \ldots a_n\$)$

LR-parsers

- Advantages
  - More powerful than LL-parsers
  - Can be generated automatically, given a (LR-) grammar
- Disadvantages
  - Hard (impossible) to construct an LR-parser manually
  - General LR parsing (Knuth, 1965) generates to big tables
    $\Rightarrow$ more restrictions are needed