Overview of Lecture 2

- Mededelingen
- Ch 2 – Language Processors
  - 2.6 Bootstrapping
  - 2.7 Triangle language processors
- Ch 3 – Compilation
  - 3.1 Phases
  - 3.2 Passes
  - 3.3 Case study: Triangle compiler
- Ch 4 – Syntactic Analysis
  - 4.1 Subphrases of syntactic analysis
  - 4.2 Grammars revisited
  - 4.3 Parsing
  - 4.4 Abstract Syntax Trees
  - 4.5 Scanning
  - 4.6 Case study: Triangle compiler

Mededelingen

- Practicum van week 1 moet (uiterlijk) aan het begin van het practicum van week 2 (d.w.z. 4 mei) afgetekend te worden.
- Practicum van week 2 is niet heel erg moeilijk, maar wel veel (van hetzelfde).
  - Bestudeer hoofdstuk 4 van W&B van te voren.

Tombstone diagrams (1)

- Tombstone diagrams
  - Set of "puzzle pieces" to reason about language processors and programs.
  - A complete diagram of a translator specifies how the source, target and implementation languages and the underlying machine are related.
  - four different kinds of pieces
  - combination rules to combine the pieces
    - not all pieces fit together
**Tombstone diagrams** (2)

- **Program** $P$ expressed in language $L$.
- **Translator** implemented in $L$, which translates programs from source language $S$ to target language $T$.
- **Interpreter** for language $M$, implemented in language $L$.
- **Machine** $M$.

---

**Bootstrapping**

- **Bootstrapping:**
  - The interpreter/compiler is implemented in the source language itself.
  - Advantage: we become less dependent on the target platform, and thus: more portable.
  - Chicken and egg problem: how do we get our first egg?
- There are several (elegant) bootstrapping schemes.

**Full Bootstrap** (1)

- A full bootstrap is needed if we need to build a new compiler from scratch.
- **Example:**
  - We build a compiler for the full Ada language.
  - We want to use Ada as the implementation language for the compiler.
  - There does not exist any Ada compiler on any machine.
  - We have a C compiler available on our machine $M$.
- **step 1:** write a compiler for a small subset of Ada in C.

**Full Bootstrap** (2)

- **step 2:** use the C-compiler to compile version v1.
- **step 3:** rewrite the Ada-S compiler in Ada-S.

---

VB HC 2 | Ch. 4 - Syntactic Analysis | 6
---

VB HC 2 | Ch. 4 - Syntactic Analysis | 7

VB HC 2 | Ch. 4 - Syntactic Analysis | 11

VB HC 2 | Ch. 4 - Syntactic Analysis | 12
**Full Bootstrap**

- step 4: use the v1-compiler to compile version v2.

- step 5: implement a compiler for full Ada in Ada-S.

**Half bootstrap**

- A half bootstrap is needed if we already have a compiler on a host machine (HM) but also want the compiler on a target machine (TM).
  - Only half of the compiler has to be rewritten: namely the codegenerator that instead of compiling to HM now has to compile to TM.

Let's write Ada → TM

We have Ada → HM

Now let us compile our new compiler with the cross compiler.

---

Ken Thompson, “Reflections on Trusting Trust”, CACM Vol. 27, No. 8, August 1984, pp. 761-763. Turing Award Lecture.

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.167.4096
**Syntactic Analysis Phase**

- **Scanner**
  - Divide the stream of characters into a stream of tokens.

- **Parser**
  - Builds the AST from the stream of tokens.

See "Basismodellen" for the theory behind scanning and parsing.

**Token stream**

Stream of tokens: whitespace and comments removed.

```plaintext
let func gcd(x: Integer, y: Integer) : Integer =
  if x // y = 0 then y
  else gcd(y, x // y);
in putint(gcd(321, 81))
```

- **Source Program**
  - `let func gcd(x: Integer, y: Integer) : Integer ~
  if x // y = 0 then y
  else gcd(y, x // y);
  in putint(gcd(321, 81))`
Tokens

- **Goal of the scanner**: translate a stream of characters to a stream of tokens.
  - Each token consists of a token type (kind) and its text representation (spelling).
  - The parser is only interested in the kind (identifier) not in the spelling (*this_is_a_very_long_identifier*).

```java
public class Token {
    private byte kind;
    private String spelling;

    public Token(byte kind, String spelling) {
        this.kind = kind;
        this.spelling = spelling;
    }
}
```

Constructing the AST

- A context free grammar generates a set of sentences.
  - Each sentence is a string of terminal symbols.
  - An (unambiguous) sentence has a unique phrase structure, embodied in its syntax tree.
  - So far, CFGs have been specified using BNF.
- EBNF (Extended BNF) = BNF + regular expressions
- Regular expressions are constructed from:
  - elements \( t \) of an alphabet \( T \)
  - symbol \( \epsilon \): the empty string
  - operators
    - choice operator \( | \)
    - concatenation operator \( \cdot \)
    - closure operator \( * \) (usually omitted)

Syntax

- Syntax is specified using “Context Free Grammars” (CFG, known from “Basismodellen”).
  - A CFG \( G \) is defined by a 4-tuple \( (N, T, P, S) \)
    - \( N \): a finite set of non-terminal symbols
    - \( T \): a finite set of terminal symbols
    - \( P \): a finite set of production rules
    - \( S \): a start symbol
- CFGs are usually written using Backus-Naur Form (BNF) notation.
- A CFG defines a set of strings, which is called the language of the CFG.
**EBNF (2)**

- Example: \( T = \{a, b, c\} \)

  \[
  \begin{align*}
  &e \rightarrow L = \{} \\
  &a \mid b \rightarrow L = \{a, b\} \\
  &b \mid c \mid a \rightarrow L = \{b, ca\} \\
  &(a \mid b)^* \rightarrow L = \{e, a, b, ab, ba, bb, bb, bbbb, \ldots\} \\
  &(b \mid c)^* \rightarrow L = \{b, c, ba, ca, baa, caa, \ldots\} \\
  \end{align*}
  \]

- Example: identifiers

  \[
  \begin{align*}
  &l ::= a \mid b \mid c \mid \ldots \mid z \rightarrow L(l) = \{a, b, c, \ldots, z\} \\
  &d ::= 0 \mid 1 \mid 2 \mid \ldots \mid 9 \rightarrow L(d) = \{0, 1, 2, \ldots, 9\} \\
  &l^*(1|d)^* \rightarrow L = \{l, ll, l1d, l1l, lld, ld1, \ldots\} = \{a, b, c, \ldots, z, a, b, a, b, a, b, a, b, a, b, a, b, \ldots\}
  \end{align*}
  \]

**Left factorization**

- Grammar transformations
  - A grammar can be transformed in a number of ways without changing the meaning. i.e. the set of strings it generates keeps the same.
  - EBNF is more flexible than BNF; it allows to perform useful transformations on grammar expressed in EBNF.

- Left factorization:
  - \( X \ Y \mid X \ Z = X (Y \mid Z) \)
  - Example:

    \[
    \begin{align*}
    \text{cmd} &:= \text{if Expr then cmd} \\
    &\mid \text{if Expr then cmd else cmd} \\
    \text{cmd} &:= \text{if Expr then cmd (e \mid else cmd)}
    \end{align*}
    \]

**EBNF (3)**

- Priorities:
  - highest: \( \ast \)
  - next: \( \cdot \)
  - lowest: \( \mid \)
  or use parentheses

- EBNF: right-hand sides of production rules are now regular expressions.

**BNF**

- Program ::= single-Command
- Command ::= single-Command
- \( \ldots \)
- Expression ::= primary-Expression
  - Expression operator primary-Expression

**Elimination of left recursion**

- Consider the left recursive production rule
  - \( N ::= X \mid NY \)
  - \( L(N) = \{X, XY, XYY, XYYY, XYYYY, \ldots\} \)

- The rule can be replaced by
  - \( N ::= X (Y)^* \)

- Example:

  \[
  \begin{align*}
  \text{Identifier} & ::= \text{Letter} \\
  &\mid \text{Identifier Letter} \\
  &\mid \text{Identifier Digit} \\
  \text{Grammar transformations not only make the grammar more concise (and more readable); they will prove very useful when constructing the parser for a grammar.}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Identifier} & ::= \text{Letter (Letter | Digit)*}
  \end{align*}
  \]
Parsing

• Terminology
  ▪ Recognition: deciding whether the input string is a sentence of the grammar $G$ or not.
  ▪ Parsing: recognition + constructing the phase structure (e.g. the concrete syntax tree).
  ▪ A grammar is unambiguous if there is only at most one way to parse any input.
    *A syntactically correct input string has a unique parse tree.*

• Two major groups of parsing algorithms
  ▪ top-down strategies
  ▪ bottom-up strategies

Micro-English

• Example – micro-English

  Sentence ::= Subject Verb Object
  Subject ::= I | a Noun | the Noun
  Object ::= me | a Noun | the Noun
  Noun ::= cat | mat | rat
  Verb ::= like | is | see | sees

  Possible sentences:

  *the cat sees a rat.*
  *I like the cat.*
  *the cat see me.*
  *I like me.*
  *a rat like me.*

Top-Down Parsing

The parser constructs the parse tree from the root node.

```
Sentence ::= Subject Verb Object
Subject ::= I | a Noun | the Noun
Object ::= me | a Noun | the Noun
Noun ::= cat | mat | rat
Verb ::= like | is | see | sees
```

Bottom-up Parsing

The parser constructs the parse tree from the bottom (terminal nodes) up (towards the root node).

```
Sentence ::= Subject Verb Object
Subject ::= I | a Noun | the Noun
Object ::= me | a Noun | the Noun
Noun ::= cat | mat | rat
Verb ::= like | is | see | sees
```

The algorithm decides here that a Noun should be an Object here and not a Subject.
Recursive-Descent Parsing (1)

- Recursive-Descent Parsing
  - straightforward top-down parsing algorithm.
  - idea: the parse tree structure corresponds to the call graph structure of the parsing procedures that call each other.
    - for each nonterminal XYZ we construct a method parseXYZ that parses this nonterminal.

- Parser for Micro-English

  Sentence ::= Subject Verb Object.

  protected void parseSentence()
  {
    parseSubject();
    parseVerb();
    parseObject();
    accept(".");
  }

  accept(t) checks if the current token is the expected token t.

Recursive-Descent Parsing (2)

Subject ::= I | a Noun | the Noun

protected void parseSubject()
{
  if (currentToken matches "I") {
    accept("I");
  } else if (currentToken matches "a") {
    accept("a");
    parseNoun();
  } else if (currentToken matches "the") {
    accept("the");
    parseNoun();
  } else
    report a syntax error
}

Given the currentToken, the method should always be able to decide which alternative to take.

Recursive-Descent Parsing (3)

public class MicroEnglishParser {
  protected Token currentToken;

  public void parse()
  {
    currentToken = first token;
    parseSentence();
    check that no token follows the sentence

    protected void accept(Token expected) { ... }
  }

  protected void parseSentence() { ... }
  protected void parseSubject() { ... }
  protected void parseVerb() { ... }
  protected void parseObject() { ... }
  protected void parseNoun() { ... }
  protected void parseVerb() { ... }
  ...}

In Watt & Brown, the parse methods are declared private. This is unfortunate as it does not allow customization of the parser through inheritance.

Recursive-Descent Parsing (4)

Systematic development of a recursive-descent parser:

1. Express the grammar in EBNF.
2. Grammar transformations:
   - eliminate left recursion
   - left-factorization
3. Create a Java Parser class with
   - protected variable currentToken
   - methods to call the scanner: accept and acceptIt
   - public method parse which
     - gets the first token from the scanner, and
     - calls the parse method of the root non-terminal of the grammar
4. Implement protected parsing methods
   - add protected methods parseN for each non-terminal N
Recursive-Descent Parsing

Consider the EBNF production rule \( N ::= \alpha \).
This production rule is converted to the parse method `parseN`.
Body of `parseN` is constructed via stepwise decomposition of \( \alpha \).

\[
\begin{align*}
\varepsilon & \quad ; \text{ (dummy statement)} \\
t & \quad \text{accept}(t) \\
P & \quad \text{parseP}() \\
P \cdot Q & \quad \text{parseP}(); \quad \text{parseQ}(); \\
P \mid Q & \quad \begin{cases} 
\text{if } (\text{currentToken in lookahead}[[N ::= P]]) & \text{parseP}(); \\
\text{else if } (\text{currentToken in lookahead}[[N ::= Q]]) & \text{parseQ}(); \\
\text{else} & \text{report a syntactic error} 
\end{cases} \\
P^* & \quad \begin{cases} 
\text{while } (\text{currentToken in lookahead}[[N ::= P]]) & \text{parseP}(); 
\end{cases}
\end{align*}
\]

The construction of a (recursive-descent) parser can be done automatically. E.g., ANTLR or javaCC

First and Follow Sets (for BNF)

\[ G = (N, T, P, S) \]

- The First set of symbols \( \alpha - \text{first}[\alpha] \) – is the set of terminals that can start a string derived from \( \alpha \)
  - For each \( \alpha \in (N \cup T)^* \): \( \text{first}[\alpha] = \{ a \mid a \in T \land \alpha \Rightarrow^* a\beta \} \cup \{ \varepsilon \mid \alpha \Rightarrow^* \varepsilon \} \)

- The Follow set of Non-Terminal \( A - \text{follow}[A] \) – is the set of terminals which may occur directly after \( A \)
  - For each \( A \in N \): \( \text{follow}[A] = \{ a \mid a \in T \land S \Rightarrow^* aA\beta \} \)

Lookahead Set (for BNF)

\[ G = (N, T, P, S) \]

- The Lookahead set of each production \( A ::= \alpha \in P \) is the set \( \text{lookahead}[[A ::= \alpha]] \) of terminals which indicate that we are in this alternative.
  - \( \text{lookahead}[[A ::= B_1B_2...B_n]] = \)
    \[ \cup \{ \text{first}(B_i)[\varepsilon] \mid 1 \leq k \leq i, B_k \Rightarrow^* \varepsilon \} \]
    \[ \cup \text{follow}[A] \quad \text{if } B_1B_2...B_n \Rightarrow^* \varepsilon \]

\[ U \]

LL(1) Grammar

\[ G = (N, T, P, S) \]

- Grammar \( G \) is LL(1), if for each pair \( A ::= \alpha, A ::= \beta \in P \) with \( \alpha \neq \beta \) it is the case that
  \[ \text{lookahead}[[A ::= \alpha]] \cap \text{lookahead}[[A ::= \beta]] = \emptyset \]

- LL(1): left-to-right, left-derivation, 1 lookahead symbol

Recursive-descent parsing only works for LL(1) grammars.
Constructing First & Follow Sets

- Initialize every $\text{first}[[N]]$ and $\text{first}[[\alpha]]$ with $\emptyset$
- Rules: add $\text{first}[[\alpha]]$ to $\text{first}[[N]]$ for every $N ::= \alpha$, with $\text{first}[[\alpha]]$ defined as:
  - $\text{first}[[\epsilon]] = \{\epsilon\}$
  - $\text{first}[[t]] = \{t\}$, $t \in T$
  - $\text{first}[[X\beta]] = \text{first}[[X]]$, if $\epsilon \notin \text{first}[[X]]$, $X \in N \cup T$
  - $\text{first}[[X\beta]] = \text{first}[[X]] \setminus \{\epsilon\} \cup \text{first}[[\beta]]$, otherwise
- Repeatedly apply rules until sets do not change anymore

Constructing First & Follow Sets

- Initialize every $\text{follow}[[N]]$ with $\emptyset$
- Rules: if there is a production rule $N ::= \alpha A \beta$, then
  - if $t \in T$ is in $\text{first}[[\beta]]$, add $t$ to $\text{follow}[[A]]$
  - if $\epsilon \in \text{first}[[\beta]]$, add $\text{follow}[[N]]$ to $\text{follow}[[A]]$
- Repeatedly apply rules until $\text{follow}$ sets do not change anymore

Recursive-Descent Parsing (8)

- Example
  - $A ::= X \text{ noot} | Y \text{ noot}$
  - $X ::= \epsilon | \text{ aap}$
  - $Y ::= \text{ mies}$
  - $\text{first}[[Y]] = \{\text{ mies}\}$
  - $\text{first}[[X]] = \{\text{ aap}, \epsilon\}$
  - $\text{first}[[A]] = \{\text{ mies, aap, noot}\}$
  - $\text{first}[[X \text{ noot}]] = \{\text{ aap, noot}\}$
  - $\text{first}[[Y \text{ noot}]] = \{\epsilon\}$
  - $\text{first}[[X]] = \{\epsilon\}$
  - $\text{first}[[Y]] = \{\text{ noot}\}$
  - $\text{first}[[A]] = \{\}$
  - $\text{lookahead}[[Y]] = \{\text{ mies}\}$
  - $\text{lookahead}[[X \text{ noot}]] = \{\text{ aap, noot}\}$
  - $\text{lookahead}[[X \text{ noot}]] = \{\epsilon\}$
  - $\text{lookahead}[[A]] = \{\}$
  - $\text{lookahead}[[X \text{ aap}]] = \{\text{ aap}\}$

- Suppose we add the following rule
  - $B ::= X \text{ aap}$
  - Now there is a problem.
  - Given $\text{ aap}$ as input in the context of $B$, we cannot decide what to do: do we take the $\epsilon$ alternative of $X$ or the $\text{ aap}$ alternative of $X$.

LL(1)

- LL(k)
  - If by looking ahead $k$ symbols in the input stream, we can always choose the right production rule, the given grammar is (strong) LL(k).
  - L: left-to-right scanning through the input stream
  - L: left-derivation

© Theo Ruys
**Parser for Mini-Triangle (1)**

Program ::= single-Command

Command ::= single-Command

single-Command ::= V-name ::= Expression

... Left-recursion

... Left-factorization needed

**Parser for Mini-Triangle (2)**

Command ::= single-Command (; single-Command)*

protected Command parseCommand() {
    parseSingleCommand();
    while (currentToken.kind == Token.SEMICOLON) {
        acceptIt();
        parseSingleCommand();
    }
}

**Parser for Mini-Triangle (3)**

single-Command ::= Identifier ( ::= Expression | ( Expression ) )

protected void parseSingleCommand() {
    switch (currentToken.kind) {
        case Token.IDENTIFIER: {
            parseIdentifier();
            switch (currentToken.kind) {
                case Token.BECOMES: {
                    acceptIt();
                    parseExpression();
                    break;
                }
                case Token.LPAREN: {
                    acceptIt();
                    parseExpression();
                    accept(Token.RPAREN);
                    break;
                }
                default: report a syntactic error
            }
        }
    }
    break;
}

... Scanning (1)

- Our parser class has two scanning-related methods:

```java
public class Parser {
    Token currentToken;

    protected void accept(byte expectedKind) {
        if (currentToken.kind == expectedKind) {
            acceptIt();
            currentToken = scanner.scan();
        } else {
            report syntax error;
        }
    }

    protected void acceptIt() {
        currentToken = scanner.scan();
    }
}
```

The purpose of scanning is to recognize the tokens in the input stream.

See Watt & Brown for more details on the parse methods for Mini-Triangle. In the laboratory of week 2 you will build your own recursive-descent parser.
• The tokens for the Triangle language are defined by the following grammar rules.

• The scanner should recognize these tokens in the input stream, and pass them to the parser.

```plaintext
Token ::= Identifier | Integer-Literal | Operator
| ::= | ::= \+-\*\/+\/\<\>=\\ | eot
Identifier ::= Letter (Letter | Digit) *
Integer-Literal ::= Digit Digit*
Operator ::= += \+- \* \+ / \< \> = \\nSeperator ::= Comment | space | eol
Comment ::= ! Graphic* eol
```

• Tasks of the scanner:
  - recognising tokens in the input stream: character string ➞ series of tokens
  - removing unwanted characters (whitespace)
  - house-keeping tasks (line numbers, listing file)
  - symbol-table management (optionally)

• Tokens are defined using regular expressions, constructed from:
  - characters
  - operators
    - concatenation (A B)
    - choice (A | B)
    - option (A?)
    - closure (A*)
  - defined regular expressions (= macros)

• ... but no recursive definitions!

• Regular expressions can be represented by transition diagrams (i.e. finite automata):
  - edges/transitions are labelled with input symbols
  - states (the nodes)
    - exactly one start state
    - any number of accepting states

Example: (a | b)c+ d

Regular Expressions and Finite Automata are equivalent.

1. Express the lexical grammar in EBNF.
2. Transcribe each EBNF production rule N::=X to a scanning method scanN, whose body is determined by X.
3. Make the scanner class consist of:
   - protected instance variable currentChar
   - protected methods take and takeIt (to advance currentChar)
   - protected scanning methods of step 2, but enhanced to record each Token's kind and spelling
   - public scan method, that returns the next token, discarding any separators and any comments.

This approach uses a recursive-descent approach to scanning. As said, usually a finite automata is constructed.

Systematic development of the scanner in W&B.
public class Scanner {
    protected char currentChar;
    protected byte currentKind;
    protected StringBuffer currentSpelling;

    public Token scan() {
        currentSpelling = new StringBuffer("");
        currentKind = scanToken();
        return new Token(currentKind, currentSpelling.toString());
    }

    protected byte scanToken() {
        switch (currentChar) {
            ...
        }
    }

    protected void take(char expectedChar) { ... }
    protected void takeIt() { ... }
    ...
}

Should have been a local variable of scan.

Append currentChar to currentSpelling and read next character into currentChar.

Syntactic Analysis Phase

Source Program

Scanner

Divide the stream of characters into a stream of tokens.

Token Stream

Parser

A token is an atomic symbol of the source program.

 Builds the AST from the stream of tokens.

Abstract Syntax Tree