Syntactic Analysis

Vertalerbouw **HC2**

Arend Rensink
kamer: Zilverling 5090
telefoon: 4862
email: rensink@cs.utwente.nl

Original design: Theo Ruys
University of Twente
Department of Computer Science
Formal Methods & Tools

What have you seen yesterday?

- Language definition consists of
  - Syntax
  - Context constraints
  - Semantics
- Syntax definition
  - Based on Context-Free Grammars
- Compilation versus interpretation
  - Many different scenarios
  - Handy notation: Tombstone diagrams

Overview of Lecture 2

- 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

HC1

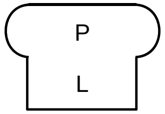
Tombstone diagrams (1)

- Tombstone diagrams
  - Set of “puzzle pieces” to reason about language processors and programs.

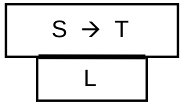
    A complete diagram of a translator specifies how the source, target and implementation languages and the underlying machine are related.
  - four different kinds of pieces
  - combination rules to combine the pieces

    not all pieces fit together

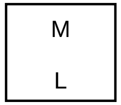
Tombstone diagrams (2)



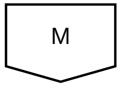
Program P expressed in language L.



Translator implemented in L, which translates programs from source language S to target language T.

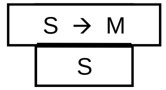


Interpreter for language M, implemented in language L.



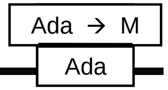
Machine M.

Bootstrapping

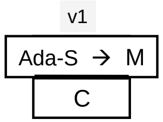


- Bootstrapping:
  - The interpreter/compiler is implemented in the source language itself.
  - Advantage: we become less dependent on the target platform, and thus: more portable.
  - Chicken and egg problem: how do we get our first egg?
- There are several (elegant) bootstrapping schemes.

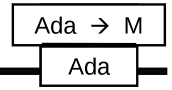
Full Bootstrap (1)



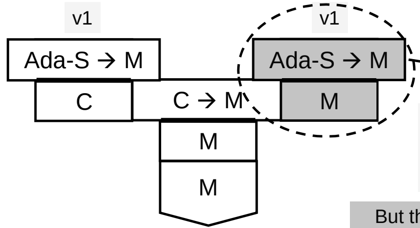
- A full bootstrap is needed if we need to build a new compiler from scratch.
- Example:
  - We build a compiler for the full Ada language.
  - We want to use Ada as the implementation language for the compiler.
  - There does not exist any Ada compiler on any machine.
  - We have a C compiler available 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.

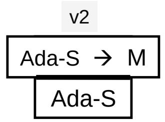


Now we can compile Ada-S programs.

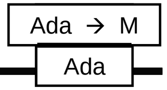
But this relies on the C-compiler being available.

- step 3: rewrite the Ada-S compiler in Ada-S.

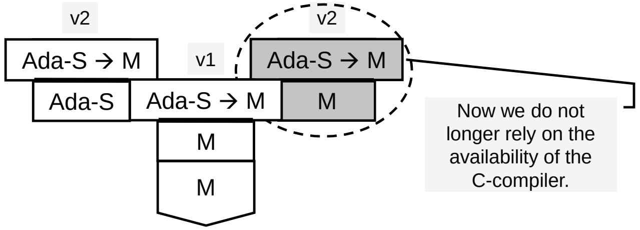
Not too difficult, given the C-implementation.



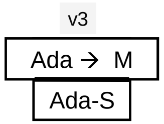
Full Bootstrap (3)



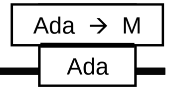
- step 4: use the v1-compiler to compile version v2.



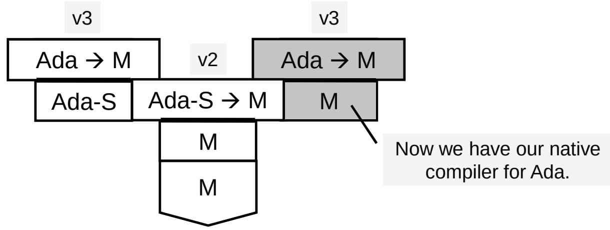
- step 5: implement a compiler for full Ada in Ada-S.



Full Bootstrap (4)



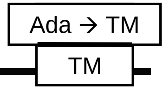
- step 6: use the v2-compiler to compile version v3.



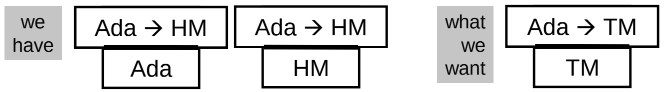
All subsequent versions of the Ada compiler can be compiled by the latest version.

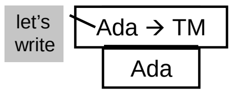
We are not longer constrained to Ada-S.

Half bootstrap (1)



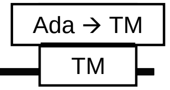
- A half bootstrap is needed if we already have a compiler on a host machine (HM) but also want the compiler on a target machine (TM).
  - Only half of the compiler has to be rewritten: namely the codegenerator that instead of compiling to HM now has to compile to TM.

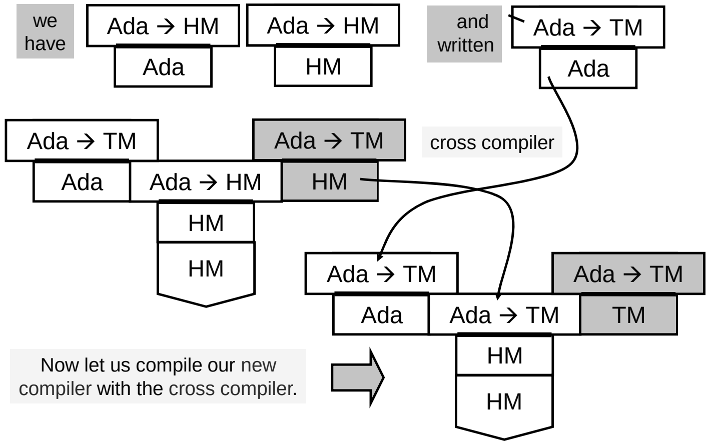




As said before, this is usually a rewrite of 50% of the compiler.

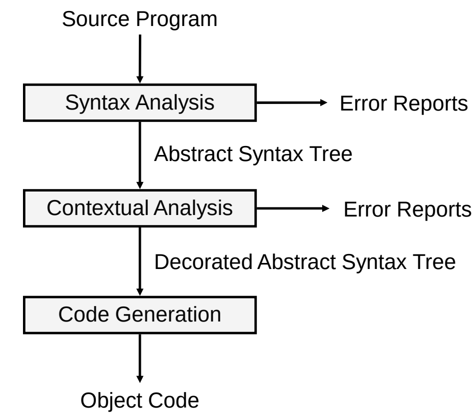
Half bootstrap (2)





Ch 3 – Compilation

Compiler Phases (1)



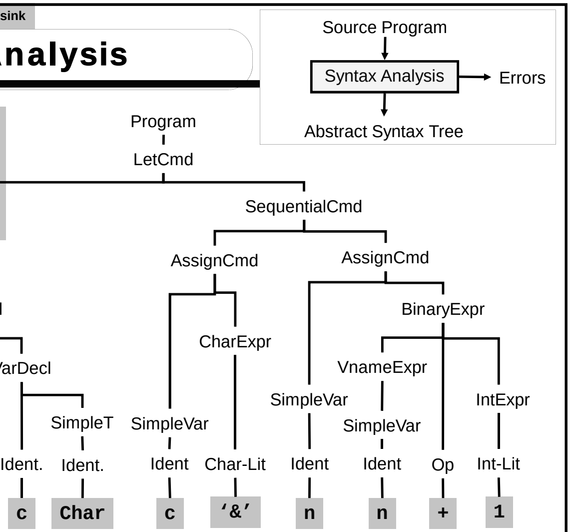
Compiler Phases (2)

- The different phases can be seen as different transformation steps to transform source code into object code.
- The different phases correspond to the different parts of the language specification:
  - Syntax analysis ↔ Syntax
  - Contextual analysis ↔ Contextual constraints
  - Code generation ↔ Semantics
- Triangle example:

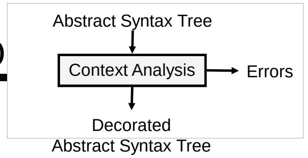
```
let
   var n: Integer;
   var c: Char
in begin
   c := '&';
   n := n+1
end
```


Syntax Analysis

```
let
   var n: Integer;
   var c: Char
in begin
   c := '&';
   n := n+1
end
```

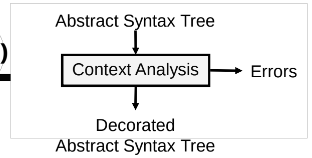


Contextual Analysis (1)

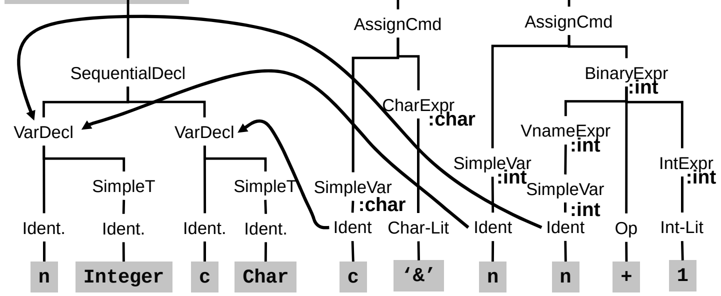


- Contextual Analysis
  - scope rules: verify that all applied occurrences of an identifier are declared
  - type rules: check whether the types of the operands of within expressions are correct
- Decorated AST
  - applied occurrences: reference to binding occurrence
  - expressions: (result) type

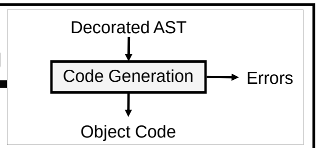
Contextual Analysis (2)




```
let
   var n: Integer;
   var c: Char
in begin
   c := '&';
   n := n+1
end
```



Code Generation



- Code Generation:
  - After syntactic and contextual analysis, it is known that the program is well-formed (or not).
  - If the source program is correct, target code is generated according to the semantics of the both languages.

Triangle source

```
let
   var n: Integer;
   var c: Char
in begin
   c := '&';
   n := n+1
end
```

TAM object code

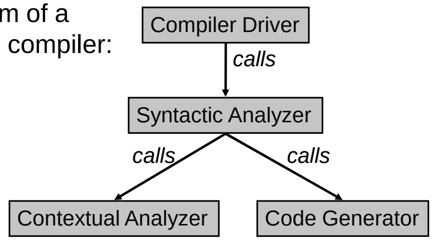
```
PUSH   2
LOADL  38
STORE  1[SB]
LOAD   0[SB]
LOADL  1
CALL   add
STORE  0[SB]
POP    2
HALT
```


Compiler passes

- A pass is a complete traversal of the source program, or a complete traversal of some internal representation of the source program.
  - A pass can correspond to a "phase" but it does not have to!
  - Sometimes a single "pass" corresponds to several phases that are interleaved in time.
- The design of a compiler is inextricably linked to the number of passes it makes.

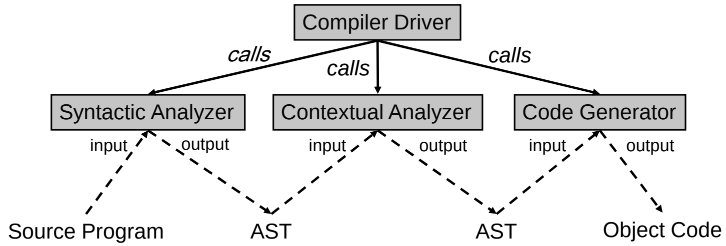
One-pass compiler

- A one-pass compiler makes a single pass over the source text, parsing, analyzing and generating code all at once.
  - Pascal compilers are usually single-pass
  - SLANG environment is single-pass.
- Structure diagram of a typical one-pass compiler:



Multi Pass Compiler

- A multi pass compiler makes several passes over the program. The output of a preceding phase is stored in a data structure and used by subsequent phases.
- Structure diagram of a typical multi-pass compiler:



Compiler Design Issues

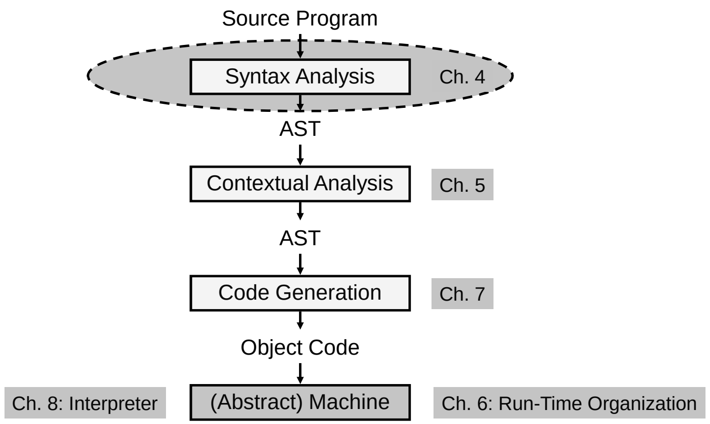
- Choice between one-pass or multi-pass is an important compiler design decision.

	one-pass	multi-pass
speed	+	-
memory	+ for large programs	+ for smaller programs
modularity	-	+
flexibility	-	+
global optimizations	- -	+
source languages	not for all PLs	

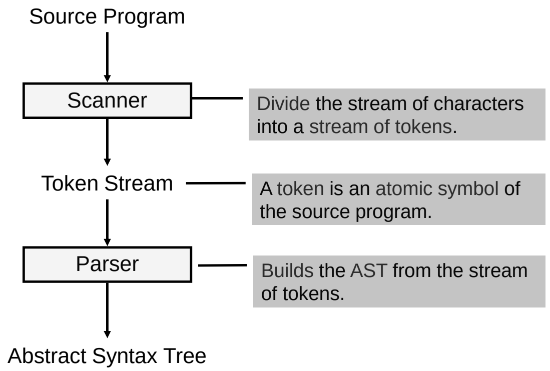
Must identifiers be declared before use?

Ch 4 – Syntactic Analysis

Compiler Phases



Syntactic Analysis Phase



Token stream

```
! Greatest Common Divider
let func gcd(x: Integer, y: Integer) : Integer ~
    if x // y = 0
    then y
    else gcd(y, x // y);
in  putint(gcd(321,81))
```

Stream of tokens: whitespace and comments removed.

`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

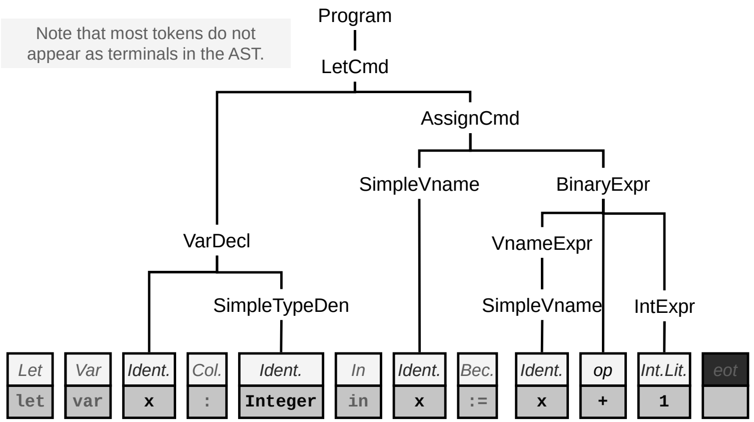
- Goal of the scanner: translate a stream of characters to a stream of tokens.
  - Each token consists of a token type (kind) and its text representation (spelling).
  - The parser is only interested in the kind (identifier) not in the spelling (**`this_is_a_very_long_identifier`**).

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

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

Each different token has a constant number.

Constructing the AST

```
let var x: Integer
in x := x+1
```



Parsing

- Terminology
  - Recognition: deciding whether the input string is a sentence of the grammar G or not.
  - Parsing: recognition + constructing the phrase 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**

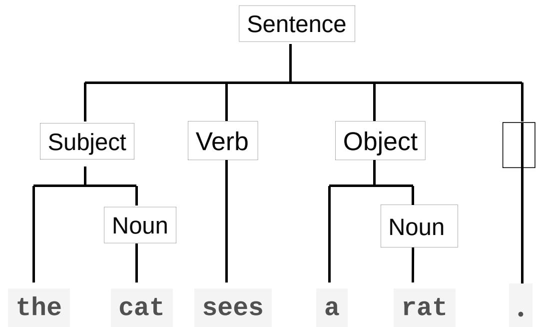
- Possible sentences:

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


Top-Down Parsing

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**

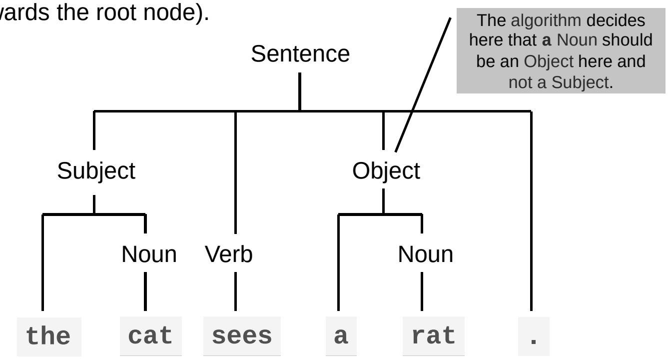
The parser constructs the parse tree from the root node.



Bottom-up Parsing

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



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.

This is only *recognition*: we do not yet *build* the AST

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 parseObject() { ... }
   protected void parseNoun() { ... }
   protected void parseVerb() { ... }
   ...
}
```

connection to the scanner which provides the tokens

Allows 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
   - perform 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
   - **`protected`** methods **`parseN`** for each non-terminal N

Recursive-Descent Parsing (5)

- 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 α.

ε	**`;`** (= dummy statement)
t	**`accept(t);`**
P	**`parseP();`**
P Q	**`parseP();`** **`parseQ();`**

P | Q	**`if (currentToken`** *in* $lookahead[N ::= P]$ **`)`**
	    `parseP();`
	`else if (currentToken` *in* $lookahead[N ::= Q]$**`)`**
	    `parseQ();`
	`else`
	    report a syntactic error

P*	**`while (currentToken`** *in* $lookahead[N ::= P]$**`)`** **`parseP();`**

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

First and Follow Sets (for BNF)

Given a CFG $G = (N, T, P, S)$

- $first[\alpha]$: the first set of string $\alpha \in (N \cup T)^*$
  - Set of terminals that can start a string derived from α
  - $first[\alpha] = \{a \mid a \in T \wedge \alpha \Rightarrow^* a\beta\} \cup \{\epsilon | \alpha \Rightarrow^* \epsilon\}$
- $follow[\alpha]$: the follow set of nonterminal $A \in N$
  - Set of terminals which may occur directly after A
  - $follow[A] = \{a \mid a \in T \ \wedge\ S \Rightarrow^* \alpha A a \beta\}$

Departure from W&B!

Lookahead Set (for BNF)

Given a CFG $G = (N, T, P, S)$

- $lookahead[A ::= \alpha]$: the lookahead set of rule $A ::= \alpha \ \in P$
  - set of terminals which indicate that we are in this alternative.
  - $lookahead[A ::= B_1 B_2 \ldots B_n] =$
    $\bigcup \{first[B_i] \setminus \{\varepsilon\} \mid i \leq n, B_1 \ldots B_{i-1} \Rightarrow^* \varepsilon\}$
    $\cup\, follow[A]$ if $B_1 \ldots B_n \Rightarrow^* \varepsilon$

LL(1) Grammar

Note: conditions on W&B page 104 are wrong.

Given a CFG $G = (N, T, P, S)$

- G is LL(1), iff
  for each pair $A ::= \alpha, A ::= \beta \in P$ with $\alpha \neq \beta$
  $lookahead[A ::= \alpha] \cap lookahead[A ::= \beta] = \emptyset$
- LL(1): left-to-right, left-derivation, 1 lookahead symbol

Recursive-descent parsing only works for LL(1) grammars.

Constructing First & Follow Sets

- Iterative construction:
  1. Initialize every $first[A]$ and $first[\alpha]$ with $\emptyset$
  2. Recompute $first[\alpha]$ according to:
     - $first[\varepsilon] = \{\varepsilon\}$
     - $first[t] = \{t\}$ if $t \in T$
     - $first[X\beta] = first[X]$, if $\varepsilon \notin first[X]$, $X \in N \cup T$
       $first[X] \setminus \{\varepsilon\} \cup first[\beta]$, otherwise
  3. Add $first[\alpha]$ to $first[A]$ for every $A ::= \alpha$
- Repeat steps 2 and 3 until sets do not change anymore

Constructing First & Follow Sets

- Iterative construction:
  1. For all $A \in N$, initialize $follow[A]$ with $\emptyset$
  2. Recompute: for all production rules $B ::= \alpha A \beta$
     - if $t \in T$ is in $first[\beta]$, add t to $follow[A]$
     - if $\varepsilon \in first[\beta]$, add $follow[B]$ to $follow[A]$
- Repeat step 2 until $follow$ sets do not change anymore

Recursive-Descent Parsing (8)

- Example

```
A ::= X noot
    | Y noot
X ::= ε
    | aap
Y ::= mies
```

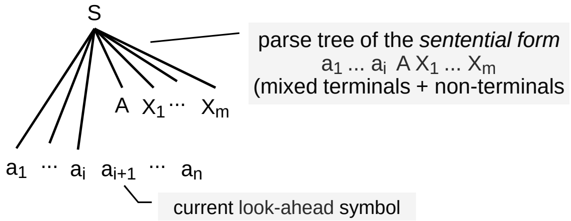
$first[Y]$ = { **mies** }
$first[X]$ = { **aap**, ε }
$first[A]$ = { **mies**, **aap**, **noot** }
$first[X$ **noot**$]$ = { **aap**, **noot** }
$first[Y$ **noot**$]$ = { **mies** }
$follow[X]$ = { **noot** }
$follow[Y]$ = { **noot** }
$follow[A]$ = { }
$lookahead[Y ::=$ **mies**$]$ = { **mies** }
$lookahead[A ::= X$ **noot**$]$ = { **aap**, **noot** }
$lookahead[A ::= Y$ **noot**$]$ = { **mies** }
$lookahead[X ::= \varepsilon]$ = { **noot** }
$lookahead[X ::=$ **aap**$]$ = { **aap** }

- Suppose we add the following rule

```
B ::= X aap
```

Now there is a problem.
Given **aap** as input in the context of B, we cannot decide what to do: do we take the ε alternative of X or the **aap** alternative of X.

LL(k)



- 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

Top-down vs. Bottom-up (1)

- Problems with top-down parsing:
  - Sometimes hard to construct a CFG which is LL(k)
  - Factorisation and elimination of left-recursion make a grammar difficult to understand
- Solution: bottom-up LR(k) parsing techniques
  - Additional advantage: more powerful than LL(k)
  - Drawback: parsing more complex and less intuitive
- LR(k)
  - L: Left-to-right scanning through the input stream
  - R: Use Right-derivation (in Reverse)
  - k: Look k symbols ahead in the input stream

Top-down vs. Bottom-up (2)

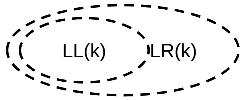
- Example:

  p_1: $A \rightarrow BB$ p_2: $A \rightarrow a$ p_3: $B \rightarrow aB$ p_4: $B \rightarrow b$

- Top-down parsing (left/right derivations)
  - begin with start symbol
  - each step: replace the left-most (or right-most) non-terminal by the right hand side of the production in the CFG.
  - continue until there is only a sentence left
  - $A \Rightarrow BB \Rightarrow BaB \Rightarrow Bab \Rightarrow aBab \Rightarrow abab$
- Bottom-up parsing
  - reverse the order of derivation: reduce a sentence to the start symbol
  - each step: replace a string that matches the right hand side of the production by the corresponding left-hand side non-terminal
  - continue until the start symbol
  - $abab \Rightarrow aBab \Rightarrow Bab \Rightarrow BaB \Rightarrow BB \Rightarrow A$

Top-down vs. Bottom-up (3)

- All LL(k) grammars are LR(k), but not the other way around



- LR(k) parsing uses the same techniques as LL(k) parsing:

  algorithms, parse tables, stacks etc.
- A LR(k) parser is essentially a Push-Down Automaton:
  - Finite number of states (a state is an abstraction of the input read)
  - Stack (with state numbers)
  - Parse table (state transitions + <u>actions</u>)
  - Always the same action for each give state + lookahead

Parser for Mini-Triangle (1)

```
Program          ::= single-Command
Command          ::= single-Command
                  |   Command ; single-Command
single-Command   ::= V-name := Expression
                  |   Identifier ( Expression )
                  |   ...
```

Left-recursion

Left-factorization needed

```
Program          ::= single-Command
Command          ::= single-Command
                         (; single-Command)*
single-Command   ::= Identifier ( := Expression
                               | ( Expression ))
                  |   ...
```


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;
      }
      ...
```

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.

Scanning (1)

- Our parser class has two scanning-related methods:

```
public class Parser {
   Token currentToken;

   protected void accept(byte expectedKind) {
      if (currentToken.kind == expectedKind)
         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.

Scanning (2)

To construct a parser, generally a parser-generator is used. But scanners are often written by hand (simple and fast).

- 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.

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


Scanning (3)

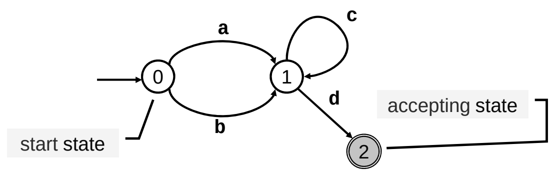
- Tasks of the scanner:
  - recognising tokens in the input stream:
    character string ⇒ series of tokens
  - removing unwanted characters (whitespace)
  - house-keeping tasks (line numbers, listing file)
  - symbol-table management (optionally)
- Tokens are defined using regular expressions, constructed from:
  - characters
  - operators
    - concatenation (A B)
    - choice (A | B)
    - option (A**?**)
    - closure (A*****)
  - defined regular expressions (= macros)
- ... but no recursive definitions!

Scanning (4)

- 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

Regular Expressions and Finite Automata are equivalent.

- Example: (**a** | **b**) **c*** **d**



RE vs CFG

- Terminals (tokens) are usually specified using regular expressions (REs).
  - a regular expression corresponds to a finite automaton
- A programming language is usually specified using a context free grammar (CFG) specified in (E)BNF.
  - a CFG corresponds to a finite automaton with a stack (a pushdown automaton)
  - a language expressed by a RE can also be expressed by a CFG (but not vice-versa).
- See [Sudkamp 1997] for details:
  - Ch. 3, 4 & 8: CFGs, parsing & pushdown automata
  - Ch. 6 & 7: REs and finite automata

Scanning (6)

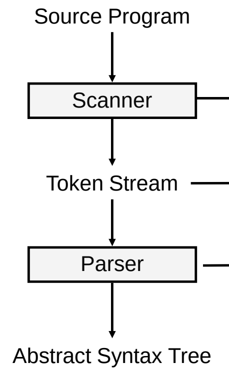
```
public class Scanner {
   protected char currentChar;
   protected byte currentKind;
   protected StringBuffer currentSpelling;

   public Token scan() {
      discard separators and whitespace;
      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**.

Given **currentChar**, a complete token is read from the input stream.

Append **currentChar** to **currentSpelling** and read next character into **currentChar**.

Syntactic Analysis Phase



Divide the stream of characters into a stream of tokens.

A token is an atomic symbol of the source program.

Builds the AST from the stream of tokens.

What have you seen today?

- More tombstones
- Compiler phases and passes
  - Syntax analysis: Scanning and parsing
  - Contextual analysis
  - Code generation
- Parsing
  - Top-down parsing: LL(k)
  - Recursive descent for LL(1)
  - Bottom-up parsing: LR(k), PDA's
- Scanning (lexical analysis)
  - Requires only regular expressions