Languages and Compilers (SProg og Oversættere)

Lecture 4
Language specifications

Bent Thomsen

Department of Computer Science

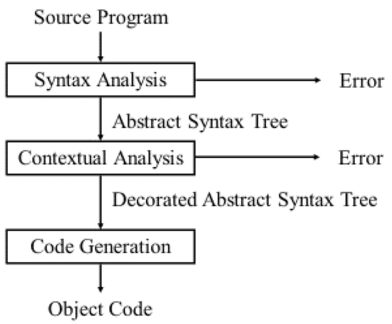
Aalborg University

Learning goals

- A deeper understanding of programming language specifications
- Introduction to context free grammars
- Introduction to BNF and EBNF
- Overview of formal specifications notations

Programming Language Specification

- Why?
  - A communication device between people who need to have a common understanding of the PL:
    - language designer, language implementor, language user
- What to specify?
  - Specify what is a 'well formed' program
    - syntax
    - contextual constraints (also called static semantics):
      - scope rules
      - type rules
  - Specify what is the meaning of (well formed) programs
    - semantics (also called runtime semantics)



Programming Language Specification

- Why?
- What to specify?
- How to specify ?
  - Formal specification: use some kind of precisely defined formalism
  - Informal specification: description in English.
  - Usually a mix of both (e.g. Java specification)
    - Syntax => formal specification using RE and CFG
    - Contextual constraints and semantics => informal
    - Formal semantics has been retrofitted though
  - But trend towards more formality (C#, Fortress)
    - fortress.pdf
    - Ecma-334.pdf

Fortress definition p. 71 and p. 181

13.4 Dotted Method Invocations

Syntax:

Primary	::=	*Primary . Id StaticArgs? ParenthesisDelimited*
ParenthesisDelimited	::=	*Parenthesized*
	|	*ArgExpr*
	|	( )
Parenthesized	::=	(*Expr*)
ArgExpr	::=	*TupleExpr*
	|	((*Expr* ,)* *Expr* ...)
TupleExpr	::=	((*Expr* ,)+ *Expr*)

A *dotted method invocation* consists of a subexpression (called the receiver expression), followed by '.', followed by an identifier, an optional list of static arguments (described in Chapter 9) and a subexpression (called the *argument expression*). Unlike in function calls (described in Section 13.6), the argument expression must be parenthesized, even if it is not a tuple. There must be no whitespace on the left-hand side of the '.' and the left-hand side of the left parenthesis of the argument expression. The receiver expression evaluates to the receiver of the invocation (bound to the self parameter (discussed in Section 10.2) of the method). A method invocation may include explicit instantiations of static parameters but most method invocations do not include them.

The receiver and arguments of a method invocation are each evaluated in parallel in a separate implicit thread (see Section 5.4). After this thread group completes normally, the body of the method is evaluated with the parameter of the method bound to the value of the argument expression (thus evaluation of the body occurs after evaluation of the receiver and arguments in dynamic program order). The value and the type of a dotted method invocation are the value and the type of the method body.

We say that methods or functions (collectively called as *functionals*) may be *applied to* (also "*invoked on*" or "*called with*") an argument. We use "call", "invocation", and "application" interchangeably.

$$[\text{R-METHOD}]\quad \frac{\texttt{object}\ O\_\ (\overline{x{:}\_})\ \_\ \texttt{end} \in p \qquad mbody_p(f[\![\overrightarrow{\tau'}]\!], O[\![\overrightarrow{\tau}]\!]) = \{(\overrightarrow{x'}) \rightarrow e\}}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).f[\![\overrightarrow{\tau'}]\!](\overrightarrow{v'})] \longrightarrow E[[\overrightarrow{v}/\overrightarrow{x}][O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})/\texttt{self}][\overrightarrow{v'}/\overrightarrow{x'}]e]}$$

The C89 standard – 519 pages

6.8 Statements and blocks

Syntax

1 *statement:*

labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement

Semantics

2 A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence.

3 A *block* allows a set of declarations and statements to be grouped into one syntactic unit. The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear.

4 A *full expression* is an expression that is not part of another expression or of a declarator. Each of the following is a full expression: an initializer; the expression in an expression statement; the controlling expression of a selection statement (**`if`** or **`switch`**); the controlling expression of a **`while`** or **`do`** statement; each of the (optional) expressions of a **`for`** statement; the (optional) expression in a **`return`** statement. The end of a full expression is a sequence point.

Forward references: expression and null statements (6.8.3), selection statements (6.8.4), iteration statements (6.8.5), the **`return`** statement (6.8.6.4).

6.8.3 Expression and null statements

Syntax

1 *expression-statement:*
  expression$_{opt}$ **;**

Semantics

2 The expression in an expression statement is evaluated as a void expression for its side effects.[134]

3 A *null statement* (consisting of just a semicolon) performs no operations.

4 EXAMPLE 1 If a function call is evaluated as an expression statement for its side effects only, the discarding of its value may be made explicit by converting the expression to a void expression by means of a cast:

```
int p(int);
/* ... */
(void)p(0);
```

134) Such as assignments, and function calls which have side effects.

5 EXAMPLE 2 In the program fragment

```
char *s;
/* ... */
while (*s++ != '\0')
        ;
```

a null statement is used to supply an empty loop body to the iteration statement.

6 EXAMPLE 3 A null statement may also be used to carry a label just before the closing } of a compound statement.

```
while (loop1) {
        /* ... */
        while (loop2) {
                /* ... */
                if (want_out)
                        goto end_loop1;
                /* ... */
        }
        /* ... */
end_loop1: ;
}
```

Forward references: iteration statements (6.8.5).

Programming Language Specification

A language specification need to address:

- Syntax
  - Token grammar: Regular Expressions
  - Context Free Grammar: BNF or EBNF
- Contextual constraints
  - Scope rules (static semantics)
    - Often informal, but can be formalized
  - Type rules (static semantics)
    - Informal or Formal
- Semantics (dynamic semantics)
  - Informal or Formal

Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

The General Problem of Describing Syntax: Terminology

- A *sentence* is a string of characters over some alphabet

- A *language* is a set of sentences

- A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`)

- A *token* is a category of lexemes (e.g., identifier)

Definition of Tokens/lexemes

- Tokens are often specified using regular expressions
- Remember:

Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	$[a-e] \mid [g-h] \mid [j-o] \mid [q-z]$
assign	"="
plus	"+"
minus	"-"
inum	$[0-9]^+$
fnum	$[0-9]^+ . [0-9]^+$
blank	$(" \ ")^+$

Figure 2.3: Formal definition of ac tokens.

Note: In most languages id is a sequence of letters and numbers starting
With a letter defined as [a-z]([a-z] | [0-9])*

Formal Definition of Languages

- **Generators**
  - A device that generates sentences of a language
  - One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator
- **Recognizers**
  - A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler

BNF and Context-Free Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages
- Backus-Naur Form (1959)
  - Invented by John Backus to describe Algol 58
  - Modified by Peter Naur to describe Algol 60
  - BNF is equivalent to context-free grammars

Syntax Specification

Syntax is specified using “Context Free Grammars”:

- A finite set of **terminal symbols** (or tokens)
- A finite set of **non-terminal symbols**
- A **start symbol**
- A finite set of **production rules**

A CFG defines a set of strings

- This is called the language of the CFG.

Backus-Naur Form

Usually CFG are written in BNF notation.

A production rule in BNF notation is written as:

$N ::= \alpha$ where N is a non terminal
and α a sequence of terminals and non-terminals

$N ::= \alpha \mid \beta \mid \ldots$ is an abbreviation for several rules with N
as left-hand side.

Sometimes non terminals are represented in angel brackets: <N> and ::= is replaced with →

Syntax Specification

Example:

```
Start ::= Letter
        | Start Letter
        | Start Digit
Letter ::= a | b | c | d | ... | z
Digit  ::= 0 | 1 | 2 | ... | 9
```

Q: What is the “language” defined by this grammar?

Note: a sequence of letters and numbers starting with a letter defined in RE as
[a-z]([a-z] | [0-9])*

What is the “language” defined by this grammar?

identifier::= available-identifier
| @ *identifier-or-keyword*
available-identifier::= identifier-or-keyword (that is not a *keyword)*
identifier-or-keyword::= identifier-start-character identifier-part-characters$_{opt}$
identifier-start-character::= letter-character
| _ (the underscore character U+005F)
identifier-part-characters::= identifier-part-character
| *identifier-part-characters identifier-part-character*
identifier-part-character::= letter-character
| *decimal-digit-character*
| *connecting-character*
| *combining-character*
| *formatting-character*
letter-character::= A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl
| A *unicode-escape-sequence* representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl
combining-character::= A Unicode character of classes Mn or Mc
| A *unicode-escape-sequence* representing a character of classes Mn or Mc
decimal-digit-character::= A Unicode character of the class Nd
| A *unicode-escape-sequence* representing a character of the class Nd
connecting-character::= A Unicode character of the class Pc
| A *unicode-escape-sequence* representing a character of the class Pc
formatting-character::= A Unicode character of the class Cf
| A *unicode-escape-sequence* representing a character of the class Cf

http://msdn.microsoft.com/en-us/library/aa664812(VS.71).aspx

What is the “language” defined by this grammar?

3.8. Identifiers

An *identifier* is an unlimited-length sequence of *Java letters* and *Java digits*, the first of which must be a *Java letter*.

```
Identifier:
  IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral

IdentifierChars:
  JavaLetter {JavaLetterOrDigit}

JavaLetter:
  any Unicode character that is a "Java letter"

JavaLetterOrDigit:
  any Unicode character that is a "Java letter-or-digit"
```

A "Java letter" is a character for which the method `Character.isJavaIdentifierStart(int)` returns true.

A "Java letter-or-digit" is a character for which the method `Character.isJavaIdentifierPart(int)` returns true.

The "Java letters" include uppercase and lowercase ASCII Latin letters `A-Z` (`\u0041-\u005a`), and `a-z` (`\u0061-\u007a`), and, for historical reasons, the ASCII underscore (`_`, or `\u005f`) and dollar sign (`$`, or `\u0024`). The `$` sign should be used only in mechanically generated source code or, rarely, to access pre-existing names on legacy systems.

The "Java digits" include the ASCII digits `0-9` (`\u0030-\u0039`).

Letters and digits may be drawn from the entire Unicode character set, which supports most writing scripts in use in the world today, including the large sets for Chinese, Japanese, and Korean. This allows programmers to use identifiers in their programs that are written in their native languages.

An identifier cannot have the same spelling (Unicode character sequence) as a keyword (§3.9), boolean literal (§3.10.3), or the null literal (§3.10.7), or a compile-time error occurs.

Two identifiers are the same only if they are identical, that is, have the same Unicode character for each letter or digit. Identifiers that have the same external appearance may yet be different.

Spot the syntax error

```
{
  x = 1;
  y = 2;
  z = 1+2
}
```

Syntax Specification

Subtle example 1:

```
Block ::= { Statements }
Statements ::= Statement ; Statements
            |  Statement

Statement  ::= V-name = Expression
            | Identifier ( Expression )
            | …
```

Syntax Specification

Subtle example 2:

```
Block ::= { Statements }
Statements ::= Statement Statements
            |  Statement

Statement  ::= V-name = Expression ;
            | Identifier ( Expression ) ;
            | …
```

Syntax Specification

Subtle example 3:

```
Block ::= { Statements }
Statements ::= Statement ; Statements
             |  Statement ;

Statement  ::= V-name = Expression
             | Identifier ( Expression )
             | …
```

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Syntax Specification

Subtle example 4:

```
Block ::= begin Statements end
Statements ::= Statement ; Statements
             |  Statement ;

Statement  ::= V-name = Expression
             | Identifier ( Expression )
             | ...
```

Syntax Specification

Bad example 4:

```
Block ::= \nl Statements \nl
Statements ::= Statement \nl Statements
             |  Statement \nl

Statement  ::= V-name = Expression
             | Identifier ( Expression )
             | ...
```

BNF Fundamentals

- In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called *nonterminal symbols,* or just *nonterminals*)
- *Terminals* are lexemes or tokens
- A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals
- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

```
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```

- Grammar: a finite non-empty set of rules
- A *start symbol* is a special element of the nonterminals of a grammar

Note: terminals/lexemes like **if** and **then** are often used in CFG instead of tokens **if_token** and **then_token**

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
<stmt> → begin <stmt_list> end
```

- Alternative rules are written with |

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

Describing Lists

- Syntactic lists are described using recursion

```
<ident_list> → ident
             | ident, <ident_list>
```

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

Pause

An Example Grammar

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const

An Example Derivation

```
<program> => <stmts> => <stmt>
                     => <var> = <expr>
                     => a = <expr>
                     => a = <term> + <term>
                     => a = <var> + <term>
                     => a = b + <term>
                     => a = b + const
```

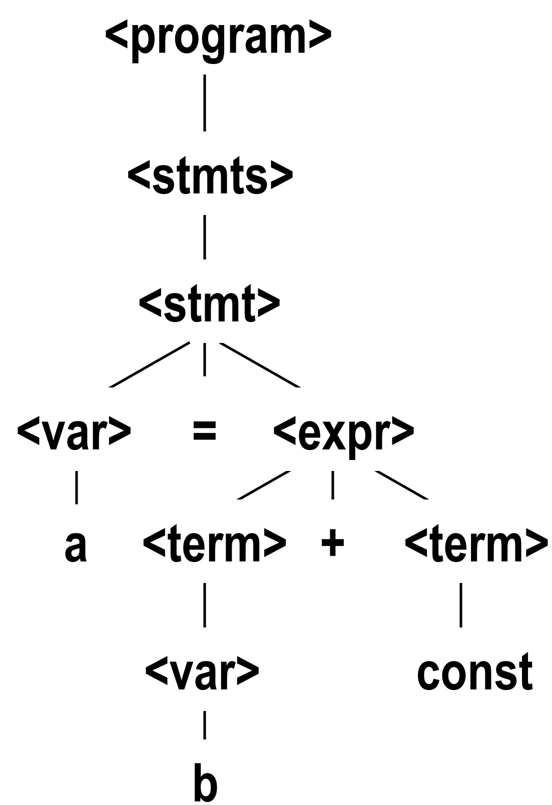
```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

Derivations

- Every string of symbols in a derivation is a *sentential form*
- A *sentence* is a sentential form that has only terminal symbols
- A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A rightmost derivation is one in which the rightmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

Parse Tree

- A hierarchical representation of a derivation

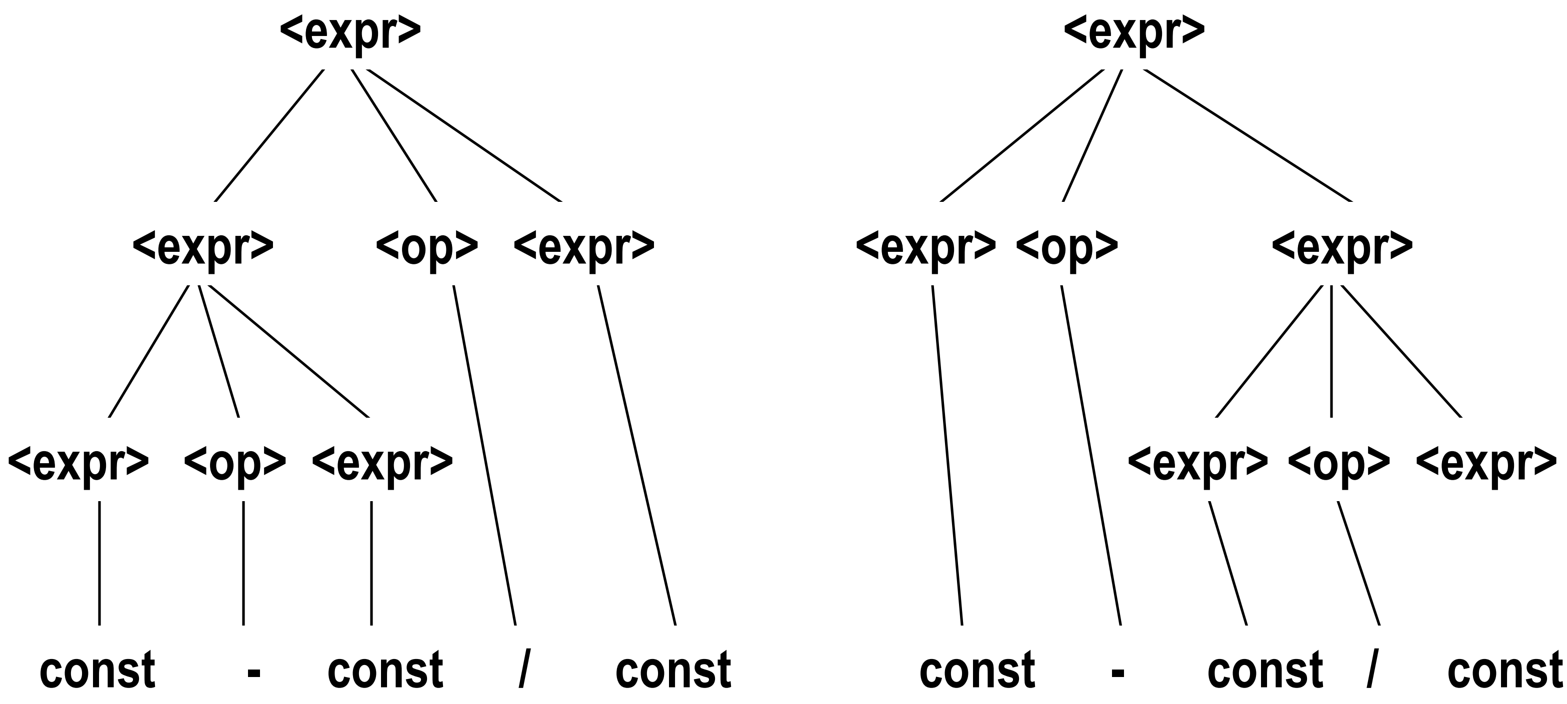


Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

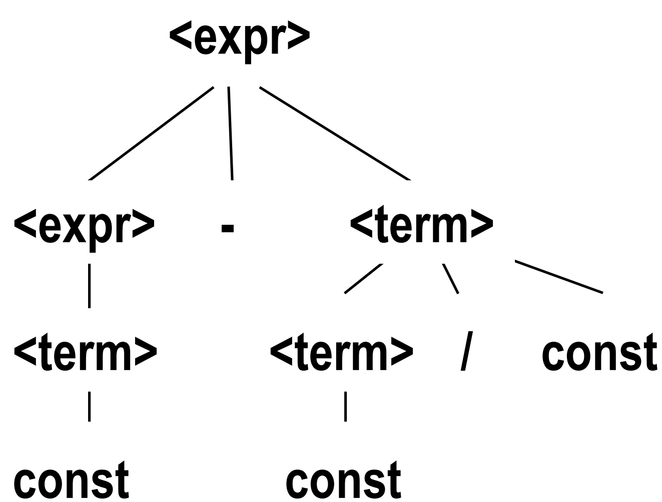
```
<expr> → <expr> <op> <expr>  |  const
<op> → /  |  -
```



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

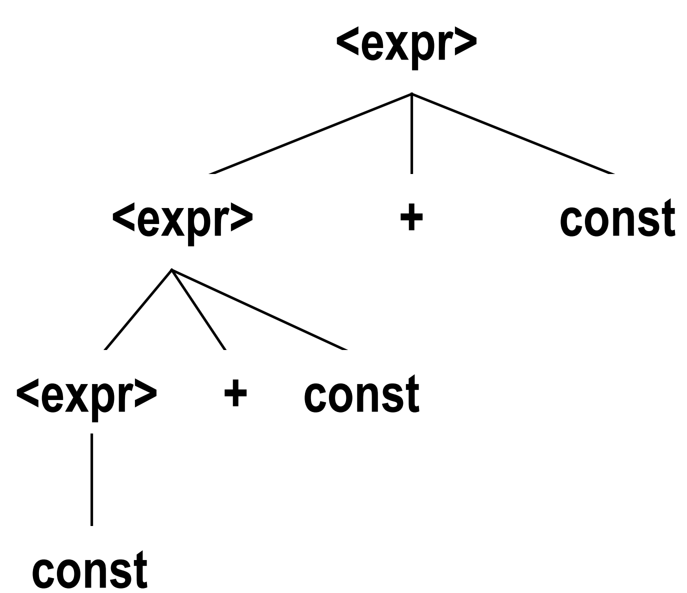
<expr> → <expr> - <term> | <term>
<term> → <term> / const| const



Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Extended BNF

- Optional parts are placed in brackets [ ]

```
<proc_call> -> ident [(<expr_list>)]
```

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

```
<term> → <term> (+|-) const
```

- Repetitions (0 or more) are placed inside braces { }

```
<ident> → letter {letter|digit}
```

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

Recent Variations in EBNF

- Alternative RHSs are put on separate lines
- Use of a colon or = or := instead of →
- Use of `opt` for optional parts
- Use of `oneof` for choices
- Sometimes terminal (lexems or tokens) are written in " " or ` ` or in **bold** or color ..
- Sometimes given in a seperate grammar and the non-terminals from this grammer is used as terminal in the CFG
- Sometimes ( )* is used for { } and ? for [ ]

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

- EBNF

```
<expr> → <term> ((+ | -) <term>)*
<term> → <factor> ((* | /) <factor>)*
```

EBNF in EBNF

```
Production    = production_name "=" [ Expression ] "." .
Expression     = Alternative { "|" Alternative } .
Alternative    = Term { Term } .
Term           = production_name | token [ "…" token ]
               | Group | Option | Repetition .
Group       = "(" Expression ")" .
Option      = "[" Expression "]" .
Repetition  = "{" Expression "}" .
```

An Example Language Specification

Mini Triangle is a very simple Pascal-like language introduced in Brown & Watt’s book: Language Processors in Java

An example program:

```
!This is a comment.
let const m ~ 7;
    var n
in
   begin
      n := 2 * m * m ;
      putint(n)
   end
```

Declarations

Expression

Command

Syntax of Mini Triangle

```
Program ::= single-Command
single-Command
      ::= V-name := Expression
        | Identifier ( Expression )
        | if Expression then single-Command
                         else single-Command
        | while Expression do single-Command
        | let Declaration in single-Command
        | begin Command end
Command ::= single-Command
          | Command ; single-Command
```

Syntax of Mini Triangle (continued)

```
Expression
   ::= primary-Expression
     | Expression Operator primary-Expression
primary-Expression
   ::= Integer-Literal
     | V-name
     | Operator primary-Expression
     | ( Expression )
V-name ::= Identifier
Identifier ::= Letter
             | Identifier Letter
             | Identifier Digit
Integer-Literal ::= Digit
                  | Integer-Literal Digit
Operator ::= + | - | * | / | < | > | =
```

Syntax of Mini Triangle (continued)

```
Declaration
  ::= single-Declaration
    | Declaration ; single-Declaration
single-Declaration
  ::= const Identifier ~ Expression
    | var Identifier := Type-denoter
Type-denoter ::= Identifier
```

```
Comment ::= ! CommentLine eol
CommentLine ::= Graphic CommentLine
Graphic ::= any printable character or space
```

Concrete Syntax of Commands

```
single-Command
        ::= V-name := Expression
          | Identifier ( Expression )
          | if Expression then single-Command
                           else single-Command
          | while Expression do single-Command
          | let Declaration in single-Command
          | begin Command end
Command ::= single-Command
          | Command ; single-Command
```

Abstract Syntax of Commands

```
Command
  ::= V-name := Expression                      AssignCmd
    | Identifier ( Expression )                 CallCmd
    | if Expression then Command
                    else Command                IfCmd
    | while Expression do Command               WhileCmd
    | let Declaration in Command                LetCmd
    | Command ; Command                         SequentialCmd
```

An abstract syntax , like the above, is often used in the definition of the formal semantics

Even more Abstract Syntax of Commands

```
Command
 ::= V-name Expression                    AssignCmd
   | Identifier Expression                CallCmd
   | Expression Command Command           IfCmd
   | Expression Command                   WhileCmd
   | Declaration Command                  LetCmd
   | Command Command                      SequentialCmd
```

An abstract syntax, like the above, may form the basis for the design of the AST

Contextual Constraints

Syntax rules alone are not enough to specify the format of well-formed programs.

Example 1:

```
let const m~2
in  m + x
```

Undefined!

Scope Rules

Example 2:

```
let const m~2 ;
    var   n:Boolean
in begin
   n := m<4;
   n := n+1
end
```

Type error!

Type Rules

Scope Rules

Scope rules regulate visibility of identifiers. They relate every **applied occurrence** of an identifier to a **binding occurrence**

Example 1

Binding occurence

```
let const m~2;
    var   r:Integer
in
    r := 10*m
```

Applied occurence

Example 2:

?

```
let const m~2
in  m + x
```

Terminology:

Static binding vs. *dynamic binding*

Type Rules

Type rules regulate the expected types of arguments and types of returned values for the operations of a language.

Examples

Type rule of **`<`** :

`E1` **`<`** *`E2`* is type correct and of type **`Boolean`**
if *`E1`* and *`E2`* are type correct and of type **`Integer`**

Type rule of **`while`**:

`while` *`E`* **`do`** *`C`* is type correct
if *`E`* of type **`Boolean`** and *`C`* type correct

Terminology:

Static typing vs. *dynamic typing*

Semantics

Specification of semantics is concerned with specifying the "meaning" of well-formed programs.

Terminology:

Expressions are **evaluated** and **yield values** (and may or may not perform side effects)

Commands are **executed** and **perform side effects**.

Declarations are **elaborated** to **produce bindings**

Side effects:

- change the values of variables
- perform input/output

Semantics

Example: The semantics of expressions.

*An **expression** is **evaluated** to yield a **value.***

An (integer literal expression) `IL` *yields the integer value of* `IL`

The (variable or constant name) expression `V` *yields the value of the variable or constant named* `V`

The (binary operation) expression `E1 O E2` *yields the value obtained by applying the binary operation* `O` *to the values yielded by (the evaluation of) expressions* `E1` *and* `E2`

etc.

Semantics

Example: The semantics of declarations.

*A **declaration** is **elaborated** to produce **bindings.** It may also have the side effect of allocating (memory for) variables.*

The constant declaration **`const`** `I~E` *is elaborated by binding the identifier value* `I` *to the value yielded by* `E`

The constant declaration **`var`** `I:T` *is elaborated by binding* `I` *to a newly allocated variable, whose initial value is undefined. The variable will be deallocated on exit from the let containing the declaration.*

The sequential declaration `D1;D2` *is elaborated by elaborating* `D1` *followed by* `D2` *combining the bindings produced by both.* `D2` *is elaborated in the environment of the sequential declaration overlaid by the bindings produced by* `D1`

Semantics

Example: The (informally specified) semantics of commands in Mini Triangle.

Commands are executed to update variables and/or perform input output.

The assignment command `V := E` *is executed as follows:*

> *first the expression* `E` *is evaluated to yield a value* ***v***
>
> *then* ***v*** *is assigned to the variable named* `V`

The sequential command `C1;C2` *is executed as follows:*

> *first the command* `C1` *is executed*
>
> *then the command* `C2` *is executed*

etc.

Structured operational semantics

$[\mathrm{ass}_{\mathrm{ns}}]$	$\langle x := a,\ s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$
$[\mathrm{skip}_{\mathrm{ns}}]$	$\langle \mathtt{skip},\ s\rangle \to s$
$[\mathrm{comp}_{\mathrm{ns}}]$	$\dfrac{\langle S_1,\ s\rangle \to s',\ \langle S_2,\ s'\rangle \to s''}{\langle S_1;S_2,\ s\rangle \to s''}$
$[\mathrm{if}^{\mathrm{tt}}_{\mathrm{ns}}]$	$\dfrac{\langle S_1,\ s\rangle \to s'}{\langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2,\ s\rangle \to s'}$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\mathrm{if}^{\mathrm{ff}}_{\mathrm{ns}}]$	$\dfrac{\langle S_2,\ s\rangle \to s'}{\langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2,\ s\rangle \to s'}$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\mathrm{while}^{\mathrm{tt}}_{\mathrm{ns}}]$	$\dfrac{\langle S,\ s\rangle \to s',\ \langle \mathtt{while}\ b\ \mathtt{do}\ S,\ s'\rangle \to s''}{\langle \mathtt{while}\ b\ \mathtt{do}\ S,\ s\rangle \to s''}$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\mathrm{while}^{\mathrm{ff}}_{\mathrm{ns}}]$	$\langle \mathtt{while}\ b\ \mathtt{do}\ S,\ s\rangle \to s$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$

Semantics

- There is no single widely acceptable notation or formalism for describing semantics
- Several needs for a methodology and notation for semantics:
  - Programmers need to know what statements mean
  - Compiler writers must know exactly what language constructs do
  - Correctness proofs would be possible
  - Compiler generators would be possible
  - Designers could detect ambiguities and inconsistencies

Semantic styles

- Structural Operational Semantics
  - Sebesta's book has a very narrow view
  - Much better view in
    - Transitions and Trees: An introduction to structural operational semantics, Cambridge University Press
- Denotational Semantics
  - Based on recursive function theory
  - Originally developed by Scott and Strachey (1970)
- Axiomatic Semantics
  - Sometimes called Hoare Logic
  - Original purpose: formal program verification

Important!

- Syntax is the visible part of a programming language
  - Programming Language designers can waste a lot of time discussing unimportant details of syntax
  - But syntax <u>is</u> important – syntax should convey the meaning intutively
- The language paradigm is the next most visible part
  - The choice of paradigm, and therefore language, depends on how humans best think about the problem
    - Imperative, Object Oriented, Functional, ..
  - There are no <u>right</u> models of computations – just different models of computations, some more suited for certain classes of problems than others
- The most invisible part is the language semantics
  - Clear semantics usually leads to simple and efficient implementations

Before Language definition

- Write programs !!
- Serves as inspiration for language specification
  - Syntax
    - Tokens
    - CFG
  - Static semantics
    - Scope rules
    - Type rules
  - Semantics
    - Informal
    - Formal
- Serves as test case for compiler !!
- Read language specifications: C, C#, Java, ..