Languages and Compilers
BNF and EBNF

Allan Milne and Adam Sampson
School of Design and Informatics
Abertay University

Today's plan

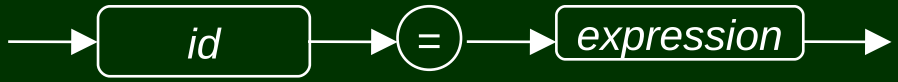
- Syntax specification
- BNF
- Derivation sequences
- EBNF

- Algol 60 reading:
    - General impressions?
    - Enough information to write your own programs?
    - Features that aren't in modern languages?
    - Missing features that you'd expect to have?

Syntax and Semantics (recap)

- A language must be defined in terms of both its ***syntax*** and ***semantics***.
- *Syntax* defines the format of the programs, statements and structures.
  - What a program looks like.
- *Semantics* defines the meaning of language constructs.
  - *What a program does.*

Specifying Syntax.

- There are a number of approaches to defining the syntax of a language:
  - Informal (e.g. many tutorials and manuals)
    - *An assignment statement has the name of an identifier followed by an equals sign and an arbitrary expression.*
  - Syntax diagrams (e.g. Pascal)
    - 

  - BNF (e.g. Algol 60)
    - <Assignment> ::= Id "=" <Expression>

Backus Normal Form (BNF).

- *Backus Naur Form* or *Backus Normal Form*.
- First used for the specification of Algol-60 – which you read last week.
- This is a <u>formal</u> specification method: there are rules for what you can say, and what exactly it means.
- Underlying formalism is *grammar theory*.
  - A grammar is a generative device that specifies how to generate valid sentences in the language.
- BNF is a <u>metalanguage</u> – a language for describing languages

BNF and languages.

- One language may be described by **many** different BNF specifications.
  - There are advantages to writing specifications in particular ways – we'll see this next week.

- A BNF specification defines only **one** language.

Naming of Parts.

metasymbols

*a **rule***

a production

<Assignment> ::= Id "=" <Expression> ;

non-terminal symbols

terminal symbol

The Parts.

- Metasymbols
  - part of the BNF language, **not** the language being specified
  - < > ::= " ; | ( ))?)+)*
- Terminal Symbols
  - atomic components of the language being specified
    (e.g. reserved words in a programming language: if, else...)
- Non-Terminal Symbols
  - define syntactic components used in the specification
  - are part of the specification not the language
  - name surrounded by <…>
  - are derived into smaller units

Productions.

- A rule in BNF can have 1 or more alternative **productions**:

<BooleanConstant> ::= true | false ;

- This rule is made up of the following two productions:

<BooleanConstant> ::= true ;

<BooleanConstant> ::= false ;

The Null Production.

- It is useful to have a notation for the null, or empty, production: a production that derives to nothing.
- We use the notation **<>** to indicate the null production.
- Note this must be an entire production
  - it cannot be used with other terminal or non-terminal symbols in a production.
- Example

<If> ::= if "(" <Condition> ")" <Statement> <Else-part> ;
<Else-part> ::= else <Statement> | <> ;

Uses Of The Null Production.

- As seen in the previous example it is used to express optionality.
- Using recursion to define repetition, the null production is often used as the bottom (or termination) of the recursion.
- Example:

```
<Bits> ::= <Bit> <Bits> | <> ;
<Bit>  ::= 0 | 1 ;
```

Using A BNF Specification.

- Applying Productions.
- Derivation Sequence.
- Languages.

Applying Productions.

- Productions of a rule are applied through substitution:

  … **<A> … → … α …**

  where the rule for <A> has a production **<A> ::= α**
- This is a <u>derivation</u> of <A>.
- Example: assume a production
  <Foo> ::= hello <Name> ;
- and a string
  a b <Foo> c d
- apply the production to get
  a b <Foo> c d → a b hello <Name> c d

Derivation Sequence.

- A **derivation sequence** is the sequence of single derivation steps starting from the **starter symbol** of the BNF and terminating in a string of terminal symbols.
- The **starter symbol** (or **distinguished symbol**) is defined to be the non-terminal of the first rule of the BNF specification.
- The final string of terminals is our program, known formally as a **sentence**.

Why Derivation Sequences?

- Why am I asking you to write down derivation sequences?
- We don't normally need to do this when we're designing compilers...
- … unless we're trying to debug a parser. As we'll see later, the derivation sequence corresponds to the series of decisions a parser has to make...
- … so you wouldn't normally write the whole thing, but you do need to be able to figure out what a bit of it should look like!

What is a language?

- The **language** defined by a BNF specification is the set of all sentences that can be derived from the starter symbol of the specification.

- Putting it another way, a sentence (or program) of a language is syntactically valid if and only if a derivation sequence can be found.

  (i.e. if there's a way to construct the sentence, using **only** the rules in the grammar)

An Example.

- Example BNF
- Example Derivation Sequence.

An Example BNF Specification.

- A specification of the syntax of a comma-separated parenthesised list of animals:

<AnimalList> ::= "(" <Animals> ")" ;

<Animals> ::= <Animal> <MoreAnimals> ;

<MoreAnimals> ::= "," <Animal> <MoreAnimals> | <> ;

<Animal> ::= ant | bat | cat | dog ;

<AnimalList> ::= "(" <Animals> ")" ;

<Animals> ::= <Animal> <MoreAnimals> ;

<MoreAnimals> ::= "," <Animal> <MoreAnimals> | <> ;

<Animal> ::= ant | bat | cat | dog ;

Here's one possible derivation sequence for “**(dog, ant)**” :

<AnimalList> → (<Animals>)
→ (<Animal> <MoreAnimals>)
→ (dog <MoreAnimals>)
→ (dog , <Animal> <MoreAnimals>)
→ (dog , ant <MoreAnimals>)
→ (dog , ant)

Also try on the board – “(cat)”, “()” (wrong)

Extended BNF.

- EBNF Specification.
- Using EBNF.
- EBNF to BNF Transformations.

Extended BNF (EBNF).

- Standard BNF relies on recursion to specify the repetition of syntactic elements.

  <Bits> ::= <Bit> <Bits> | <> ;
  <Bit> ::= 0 | 1 ;

- In many languages, it's simpler and more concise to use an **iterative** construct to describe repetition.

- Extended BNF (EBNF) adds iterative constructs to the standard BNF metalanguage.

- BNF is a **subset** of EBNF.

EBNF Clauses.

- EBNF adds the following clauses that can be included anywhere in a production:
  - **(α | β | …)** bracket - occurs exactly 1 time
  - **(α | β | …)?** optional - occurs 0 or 1 time
  - **(α | β | …)*** occurs 0 or more times
  - **(α | β | …)+** occurs 1 or more times
- Example

<Number> ::= (+|-)? (<Digit>)+ (. (<Digit>)*)? ;
<Digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ;

Using EBNF.

- EBNF clauses can be transformed into standard BNF with recursive rather than iterative productions (next slide).
- Using EBNF allows many alternative, but equivalent, BNF specifications:

<AnimalList> ::= "(" <Animals> ")" ;
<Animals> ::= <Animal> <MoreAnimals> ;
<MoreAnimals> ::= , <Animal> <MoreAnimals> | <> ;
<Animal> ::= ant | bat | cat | dog ;

<AnimalList> ::= "(" <Animal> <MoreAnimals> ")" ;
<MoreAnimals> ::= (, <Animal>)* ;
<Animal> ::= ant | bat | cat | dog ;

<AnimalList> ::= "(" <Animal> (, <Animal>)* ")" ;
<Animal> ::= ant | bat | cat | dog ;

EBNF to BNF Transformation Rules

$<A> ::= \delta_1 \ (\alpha \mid \beta \mid \ldots) \ \delta_2 \ ;$	$\rightarrow$	$<A> ::= \delta_1 \ <X> \ \delta_2 \ ;$ $<X> ::= \alpha \mid \beta \mid \ldots \ ;$
$<A> ::= \delta_1 \ (\alpha \mid \beta \mid \ldots)? \ \delta_2 \ ;$	$\rightarrow$	$<A> ::= \delta_1 \ <X> \ \delta_2 \ ;$ $<X> ::= \alpha \mid \beta \mid \ldots \mid <> \ ;$
$<A> ::= \delta_1 \ (\alpha \mid \beta \mid \ldots)^* \ \delta_2 \ ;$	$\rightarrow$	$<A> ::= \delta_1 \ <Y> \ \delta_2 \ ;$ $<X> ::= \alpha \mid \beta \mid \ldots \ ;$ $<Y> ::= <X> \ <Y> \mid <> \ ;$
$<A> ::= \delta_1 \ (\alpha \mid \beta \mid \ldots)+ \ \delta_2 \ ;$	$\rightarrow$	$<A> ::= \delta_1 \ <X><Y> \ \delta_2 \ ;$ $<X> ::= \alpha \mid \beta \mid \ldots \ ;$ $<Y> ::= <X> \ <Y> \mid <> \ ;$

EBNF Derivation Sequences

- We can write derivation sequences for EBNF too...
- Remember to make **one** decision at each step!
  - Expand **one** element, based on what it might match in the input
- For example...

<AnimalList> ::= "(" <Animal> (, <Animal>)* ")" ;

<Animal> ::= ant | bat | cat | dog ;

The EBNF derivation sequence for “**(dog, ant)**” :

<AnimalList> → (<Animal> (, <Animal>)*)
→ (dog (, <Animal>)*)
→ (dog , <Animal> (, <Animal>)*)
→ (dog , ant (, <Animal>)*)
→ (dog , ant)

Note how we expanded **(...)*** –
the first time, **(foo)*** expanded to **foo (foo)***
(because foo matched)
the second time, **(foo)*** expanded to nothing
(because foo didn't match)

<AnimalSet> ::= "<" (ant | bat | cat | dog)+ ">" ;

The EBNF derivation sequence for “**< dog ant >**” :

<AnimalSet> → < (ant | bat | cat | dog)+ >

→ < (ant | bat | cat | dog) (ant | bat | cat | dog)* >

→ < dog (ant | bat | cat | dog)* >

→ < dog (ant | bat | cat | dog) (ant | bat | cat | dog)* >

→ < dog ant (ant | bat | cat | dog)* >

→ < dog ant >

Note how we expanded **(foo)+**...

the first time, **(foo)+** became **foo (foo)***

i.e. one or more foos → one foo, then zero or more foos

Don't forget to write out the choice (ant | bat | …) before deciding

Any questions?

- **Practical**: some exercises with BNF and EBNF:
  - test whether a sentence is valid
  - produce a derivation sequence
- I'll put answers to these up next week

- **Next lecture**: properties of BNF specifications, and how to explain them to a parser...