Introduction

- Primitives of languages (tokens)
  - Keywords, operators, identifiers

  How are they specified?
- How are strings of programming languages specified?
  - Meta-notation for describing strings of programming languages (programs)
  - Three different notations: BNF, EBNF, Syntax diagram (SD)
  - How do BNF, EBNF, and SD differ?
  - How can we convert from one to another?
- How are strings recognized?
  - Does a string belong to a programming language?
  - If so, how was it constructed?

  Use syntax to derive and understand what string is.
- Recursive descent parser: how can EBNF/BNF rules can be systematically converted into a parser?
- Activities:
  - Homework
  - Project

What characterizes a programming language?

Syntax

- Symbols: `if, when, while, for, identifiers`
- Rules for combining symbols:

```
if <cond> then <statement>
```

  Rules determine: "does this string belong to language?"
- Analyzed by the compiler to determine how symbols are combined. Also detect syntactical errors:

  Examples:

```
a := b + c; (okay)
foo)a, b(; (does not look right)
```

- Syntax usually represented using a formalism, called *Context Free Grammars (CFG).*

  Our focus: BNF, EBNF and Syntax Diagrams

Semantics

- What does a specific sequence of symbols mean?
- Used by compiler to implement a language
- Semantic errors:

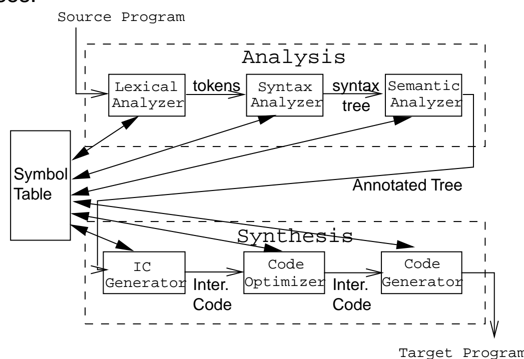
  Examples:

```
X := 5; // X is undefined
a[x] := 5; // x outside range?
```

- Many semantic errors cannot be recognized automatically using CFG recognizers.

Compilation

- Phases:



- Lexical analysis (Scanner)
  - Discard white spaces and comments
  - Group a sequence of characters into tokens (identifiers, numbers, operators, keywords, etc.)
  - tokens: primitives of language
- Syntactic analysis (Parser)
  - Process of recognizing strings (sentences) in a language
  - Identify and represent syntactic structure of a string
  - Used for semantic analysis and code generation
  - Many ways: will look at a simple method (ECS 142)
- Semantic analysis — type checking
- Code generation
- Code optimization

Syntax: Backus-Naur Form (BNF)

- A meta-language for describing a programming language

  First used in Algol60 report.
- Has four parts:
  1. *Atomic symbols*: A set of terminals, tokens, primitives
  2. *Syntactic categories*: A set of non-terminals (NT)

     Each NT denotes a set of strings
  3. *Productions*: a set of rules
  4. *Starting non-terminal*: a nonterminal used to derive strings

     Set of strings associated with starting nonterminal represents language.
- BNF uses following notations:

  (i) Non-terminals enclosed in $<$ and $>$.

  (ii) Rules written as

$$X ::= Y$$

  (1) X is LHS of rule and can only be a NT.

  (2) Y is RHS of rule: Y can be

  (a) a terminal, nonterminal, or concatenation of terminal and nonterminals, or

  (b) a set of strings separated by alternation symbol $|$.

  Example:

$$< S > ::= a < S > | a$$

- Notation: ϵ: Used to represent an empty string (a string of length 0).

BNF - cont'd.

- **Convention**: Specify grammar by listing production rules; list the rules for starting NT first.
- A grammar derives a string by
  - beginning with starting NT;
  - repeatedly replacing a NT by a right hand production for that NT until no more NTs remain.

  Note: Order of replacement does not matter
- A language $L(G)$ associated with grammar G is the set of all strings derived from the starting NT of G.
- Example:
  - Terminals: A, B, $\cdots$Z; 0, 1, $\cdots$ 9
  - Nonterminals: <id>, <rest>, <alpha>, <alphanum>, <digit>
  - Starting NT: <id>
  - Productions/rules:

```
<id>      ::= <alpha> | <alpha> <rest>
<rest>    ::= <rest> <alphanum> | <alphanum>
<alphanum>::= <alpha> | <digit>
<alpha>   ::= A | B | ··· | Z
<digit>   ::= 0 | 1 | ··· | 9
```

  - <id> is either <alpha>, or <alpha> followed by <rest>.
  - Do the following belong to the language? A, B2D, 2A, A*BL
  - What is the language?

Parse tree

- Represent syntactic structure of a string. It describes graphically how a string is generated from rules
  - Root: starting NT
  - Interior nodes: NTs
  - Leaves: terminals/atoms
  - Edge from one node b to nodes $a_1, a_2, \cdots, a_n$ if rule of form:

$$b ::= a_1 a_2 \cdots a_n$$

- Parsers create parse tree to create a representation of an input source program. Parse tree (or some representation of it) used as a basis for semantic analysis and code generation (called syntax-directed translation) $\Longrightarrow$ an important data structure
- Condensed version of tree: abstract syntax tree.
- An example BNF for defining expressions in a language:

```
<exp>    ::=<exp> + <term> | <term>
<term>   ::=<term> * <term>
              | '('<exp>')'
              | <number>
<number> ::= 0 | 1 | ··· | 9
```

- Show
  - a parse tree for expression 2 + (3 * 4);
  - Derivation and corresponding creation of parse tree

Parse tree - cont'd.

- Parse trees can be used to evaluate expressions: Bottom up evaluation for 2 + 3 * 4

  (Show parse tree)
- Note that some of the semantics associated with expressions are already specified by the way grammar is written.
- Precedence of * over +: by deriving * lower in the parse tree.
- Left recursion

  `<exp> ::= <exp> + <term>`

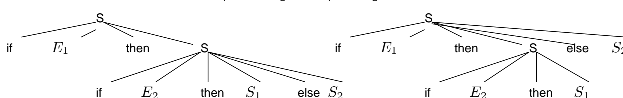
  left associativity of +
- Right recursion:

  `<exp> ::= <term> + <exp>`

  right associativity of +
- Ambiguity: More than one parse tree for same string $\Longrightarrow$ grammar is ambiguous.

```
S ::= if E then S
S ::= if E then S else S
```

if E_1 then if E_2 then S_1 else S_2



Which one to choose?

  - Either change grammar, or
  - Define rules for selecting one

Extended BNF (EBNF)

- EBNF: adding more meta-notation $\Longrightarrow$ shorter productions
- NTS begin with uppercase letters (discard <>)
- Terminals that are grammar symbols ('[' for instance) are enclosed in quotes ('').

```
Sq        ::= {'[' ']'}
```

- Repetitions (zero or more) are enclosed in {}

  Rewrite the previous grammar:

```
Alpha    ::= A | B | ... | Z
Digit    ::= 0 | 1 | ... | 9
Id       ::= Alpha { Alphanum }
Alphanum ::= Alpha | Digit
```

- Zero or one (options) are enclosed in [ ]:

```
Ifstmt ::= if Cond then Stmt |
           if Cond then Stmt else Stmt
```

  Rewrite as:

```
Ifstmt ::= if Cond then Stmt [else Stmt]
```

- Use () to group items together:

```
Exp ::=   Item {+ Item}
        | Item {- Item}
```

  Rewrite as

```
Exp ::= Item {(+|-) Item}
```

Conversion from EBNF to BNF and Vice Versa

- **BNF to EBNF:**

  (i) Look for recursion in grammar:

  `A ::= a A | B`

  rewrite as:

  `A ::= { a } B`

  (ii) Look for common string that can be factored out with grouping and options.

  `A ::= a B | a`

  Rewrite as:

  `A := a [B]`

- **EBNF to BNF:**

  (i) Options: [ ]

  `A ::= a [B] C`

  Rewrite as:

  ```
  A' ::= a N C
  N ::= B | empty string
  ```

  (ii) Repetition: { }

  `A ::= a { B1 B2 ... Bn } C`

  Rewrite as:

  ```
  A' ::= a N C
  N  ::= B1 B2 ... Bn N | empty string
  ```

  (iii) Grouping: ( )

Syntax Graphs/Diagrams

- Graphical representation of grammar rules
- Direct mapping to/from EBNF

nonterminal	Y	→ Y →
terminal	x	→ (x) →
sequence	Y1 Y2	→ Y1 → Y2 →
alternation	Y1 | Y2 | Y3	Y1 / Y2 / Y3
optional	[Y]	Y
repetition	{ Y }	Y

Example

- EBNF:

```
Program    ::= Block
Block      ::= { Stmt }
Stmt       ::= Assgn | While
Assgn      ::= Id ':=' Exp
Exp        ::= Id | Num
While      ::= while Exp do Block end
```

Nonterminal `Num` represents nonempty sequence of digits and Id represents a nonempty sequence of letters. Note though that we treat Id and Num as terminals because they are tokens returned by the scanner.

- Example string:

```
while id do
  id := 3
end
```

- **Give syntax graph**

Constructing a Parser from EBNF and Syntax Graph

- What does a parser do?
- Recursive descent

  Also called *predictive parser* (Parser commits itself to a rule based on a single symbol)

- Approach:
  - Determine "first" sets
  - Translate syntax and first sets into code

First

- first(V) = Set of all terminals that can *begin* a string derived from V, and ϵ if ϵ in V.

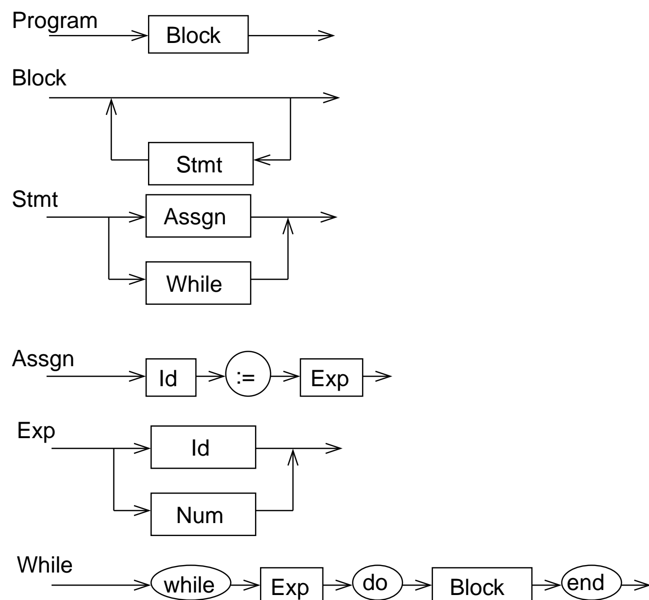
  ϵ: empty string (string of length 0)

- Example (grammar on slide 9):
  - `first(While) = { while }`
  - `first(Exp) = { Id, Num }`
  - `first(Assgn) = { Id }`
  - `first(Stmt) = first(Assgn)` ∪ `first(While) = { Id, while }`
  - `first(Block) = first(Stmt)` ∪ { ϵ } = { `Id, while,` ϵ }
  - `first(Program) = first(Block) = { Id, while,` ϵ }

Example

- EBNF and syntax graph:

```
Program    ::= Block
Block      ::= { Stmt }
Stmt       ::= Assgn | While
Assgn      ::= Id ':=' Exp
Exp        ::= Id | Num
While      ::= while Exp do Block end
```



Parser construction

- Assume:
  - `sym`: global variable to store a token
  - `next`: routine that sets `sym` to token in input
  - `f_X`: denotes `first(X)`
  - `error`: error handling routine
- Parser construction:
  - Construct one procedure for each nonterminal procedure `X()` for nonterminal `X`
  - Terminal: `x`
    ```
    if (sym is terminal x)
       next();
    else error();
    ```
  - Nonterminal: `X`
    ```
    X();
    ```
  - Sequence: `X1 X2`
    ```
    X1(); X2()
    ```
  - Alternation: `X1 | X2`
    ```
    if (sym ∈ f_X1) X1();
    else if (sym ∈ f_X2) X2();
    else error();
    ```
  - Repetition: `{ X1 }`
    ```
    while (sym ∈ f_X1)
       X1();
    ```


Pseudo-C code for parsing example grammar

```
main() {
   /* read the first token */
   next();
   /* parse the input */
   Program(); /* starting nonterminal */
   /* do something to ensure that
   all input was parsed */
}

Program() {
   Block();
}

Block() {
   while (sym ∈ f_Stmt)
      Stmt();
}

Stmt() {
   if (sym ∈ f_Assgn)
      Assgn();
   else if (sym ∈ f_While)
      While();
   else error();
}
```



```
Assgn() {
   if (sym is an Id)
      next();
   else error();
   if (sym is a ':=')
      next();
   else error();
   Exp();
}

Exp() {
   if (sym is an Id)
      next();
   else if (sym is a Num)
      next();
   else error();
}

While() {
   if (sym is a while)
      next();
   else error();
   Exp();
   if (sym is a do)
      next();
   else error();
   Block();
   if (sym is an end)
      next();
   else error();
}
```