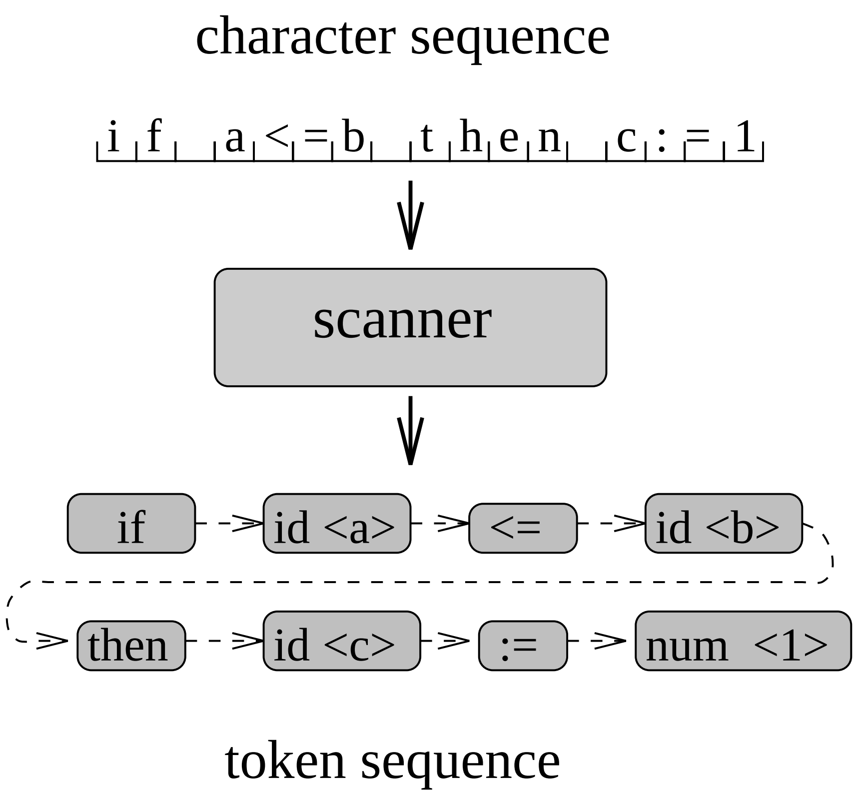
Class Information

- Special permission numbers: last call. Today is deadline for adding a course.
- Second homework will be posted on Tuesday.

Review: Lexical Analysis (Scott 2.1, 2.2)



Tokens (Terminal Symbols of CFG, Words of Lang.)

- Smallest "atomic" units of syntax
- Used to build all the other constructs
- Example, Pascal:

  keywords: `program begin if then` ...

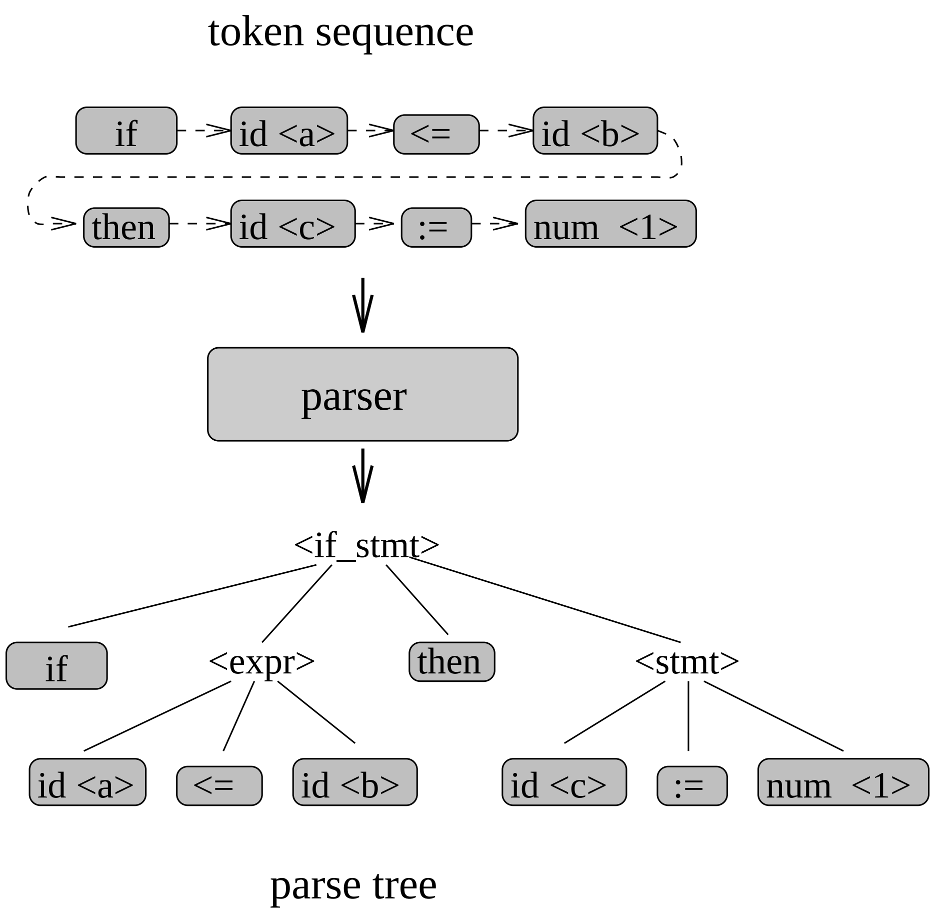
  `= * / - < > = <= >= <>`

  `( ) [ ] ; := . ,` ...

  `number` (Example: `3.14 28` ...)

  `identifier` (Example: `b square addEntry` ...)

Syntax Analysis (Scott, Chapter 2.3)



BNF (Backus-Naur Form): A formal notation for describing syntax— how components can be combined to form a valid program.

- To specify which programs are legal
- To describe the structure of programs (*parse tree*)
- BNF is a way of writing context free grammars (CFGs)

Context Free Grammars (CFGs)

- A formalism for describing languages
- A CFG $\mathcal{G}$ is a quadruple $\mathcal{G} =< T, N, P, S >$:
  1. A set T of terminal symbols (tokens)
  2. A set N of nonterminal symbols
  3. A set P production (rewrite) rules
  4. A special start symbol S
- The language $L(\mathcal{G})$ is the set of sentences of terminal symbols in T^* that can be derived from the start symbol S: $L(\mathcal{G}) = \{w \in T^* | S \Rightarrow^* w\}$

CFGs are rewrite systems with restrictions on the form of rewrite (production) rules that can be used

A partial example of a CFG in BNF:

...

<if-stmt> ::= **if** <expr> **then** <stmt>
<expr> ::= **id** <= **id**
<stmt> ::= **id** := **num**

Elements of BNF Syntax

Terminal Symbol: **Symbol-In-Boldface**

Non-Terminal Symbol: *Symbol-In-Angle-Brackets*

Production Rule:

Non-Terminal ::= Sequence of Symbols

or

Non-Terminal ::= Sequence | Sequence | ...

Alternative Symbol: |

Empty String: ϵ

How a BNF Grammar Describes a Language

- A *sentence* is a sequence of terminal symbols (tokens)
- The language $L(\mathcal{G})$ of a BNF grammar $\mathcal{G}$ is the set of sentences generated using the grammar:
  - Begin with start symbol.
  - Iteratively replace non-terminals with terminals according to rules (rewrite system).
  - This replacing process is called a **derivation** ($\Rightarrow$). Zero or multiple derivation steps are written as $\Rightarrow^*$.
  - formally: $L(\mathcal{G}) = \{w \in T^* | S \Rightarrow^* w\}$

Simple BNF Grammar ($\mathcal{G}$)

Terminals letters, digits, **:=**

Nonterminals <letter> <digit> <identifier> <stmt>

Productions

1. <letter> ::= A | B | C | ... | Z
2. <digit> ::= 0 | 1 | 2 | ... | 9
3. <identifier> ::= <letter> |
   <identifier> <letter> |
   <identifier> <digit>
4. <stmt> ::= <identifier> := 0

Start Symbol <stmt>

Derivation in a Grammar ($\mathcal{G}$)

Is `X2 := 0` $\in L(\mathcal{G})$, i.e., can `X2 := 0` be derived in $\mathcal{G}$?

In which order to apply the rules?

Typically, there are three options:

leftmost ($\Rightarrow_L$)

rightmost($\Rightarrow_R$)

any ($\Rightarrow$)

Does it matter?

Derivation in a Grammar ($\mathcal{G}$)

Is `X2 := 0` $\in L(\mathcal{G})$, i.e., can `X2 := 0` be derived in $\mathcal{G}$?

leftmost derivation		rule applied
<stmt>	$\Rightarrow_L$	4
<identifier> := 0	$\Rightarrow_L$	3c
<identifier><digit> := 0	$\Rightarrow_L$	3a
<letter><digit> := 0	$\Rightarrow_L$	1
X <digit> := 0	$\Rightarrow_L$	2
X2 := 0		

rightmost derivation		rule applied
<stmt>	$\Rightarrow_R$	4
<identifier> := 0	$\Rightarrow_R$	3c
<identifier><digit> := 0	$\Rightarrow_R$	2
<identifier> 0 := 0	$\Rightarrow_R$	3a
<letter> 0 := 0	$\Rightarrow_R$	1
X2 := 0		

Parsing of a language L($\mathcal{G}$)

Can we **recognize** `X2 := 0` as being in L($\mathcal{G}$)?

X2 := 0	
<letter>2 := 0	1
<identifier>2 := 0	3a
<identifier><digit> := 0	2
<identifier> := 0	3c
<stmt>	4

Note: Different parsing techniques, i.e., the automatic recognition sentences $w \in L(G)$ will be discussed in more detail in `198:415 Compilers`.

We will talk about LL(1) grammars and an example parser for a small language (tinyL) that is implemented using mutually recursive procedures (recursive descent parser).

Parse Trees (in $\mathcal{G}$)

A *parse tree* of `X2 := 0` in $\mathcal{G}$:

Each internal node is a nonterminal; its children are the RHS of a production for that NT.

The parse tree demonstrates that the grammar generates the terminal string on the frontier.

A Language May Have Many Grammars

Consider $\mathcal{G}'$:

Terminals letters, digits, **:=**

Nonterminals <letter> <digit> <ident> <stmt> <letterordigit>

Productions

1. <letter> ::= A | B | C | ... | Z
2. <digit> ::= 0 | 1 | 2 | ... | 9
3. <ident> ::= <letter> |
   <ident> <letterordigit>
4. <stmt> ::= <ident> := 0
5. <letterordigit> ::= <letter> | <digit>

Start Symbol <stmt>

A Language May Have Many Grammars

$\mathcal{G}$ and $\mathcal{G}'$ generate the same language, but yield different parse trees.

Example: A *parse tree* of `X2 := 0` in $\mathcal{G}'$.

Grammars and Programming Languages

Many grammars may correspond to one programming language.

Good grammars:

- capture the logical structure of the language
  $\Rightarrow$ structure carries some semantic information
  (example: expression grammar)
- use meaningful names
- are easy to read,
- are unambiguous
- ...

What's the problem with ambiguity?

Ambiguous Grammars

"Time flies like an arrow; fruit flies like a banana."

A grammar $\mathcal{G}$ is ambiguous iff there exist a $w \in L(\mathcal{G})$ such that there are

1. two distinct parse trees for w, or
2. two distinct leftmost derivations for w, or
3. two distinct rightmost derivations for w.

We want a unique semantics of our programs, which typically requires a unique syntactic structure.

Simple Statement Grammar

<start> ::= <stmt>

<stmt> ::= <if-stmt> | <assgn>

<if-stmt> ::= **if** <expr> **then** <stmt> |
if <expr> **then** <stmt> **else** <stmt>

<assgn> ::= <id> := <d>

<expr> ::= <id> = 0

<d> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<id> ::= a | b | c | . . . | z

Dangling Else Ambiguity

How are nested **if** statements parsed with this grammar?

if x = 0 **then** **if** y = 0 **then** z := 1 **else** z := 2

Dangling Else Ambiguity

if x = 0 **then if** y = 0 **then** z := 1 **else** z := 2

How to deal with ambiguity?

1. Change the language to include **delimiters** (e.g.: new terminal symbol)
   Examples: dangling else, expression grammar
2. Change the grammar
   Example: impose **associativity** and **precedence** in an arithmetic expression grammar

Changing the Language to Include Delimiters

Algol 68 if statement:

<if-stmt> ::= **if** <expr> **then** <stmt> **fi** |
if <expr> **then** <stmt>
else <stmt>
fi

How would you use this syntax to express the meaning of the two different parse trees for:

if x = 0 **then** **if** y = 0 **then** z := 1 **else** z := 2

Arithmetic Expression Grammar

<start> ::= <expr>

<expr> :: = <expr> + <expr> |

<expr> - <expr> |

<expr> * <expr> |

<expr> / <expr> |

<expr> ^ <expr>|

<d> | <l>

<d> :: = 0 | 1 | 2 | 3 | . . . | 9

<l> :: = a | b | c | . . . | z

Possible Parse Trees

Parse “$8 - 3 * 2$”:

Changing the Language to Include Delimiters

<expr> ::= (<expr>) − (<expr>) |

(<expr>) ∗ (<expr>) |

<l> | <d>

```
(8)-((5)*(2))
((8)-(5))*(2)
```

Pretty ugly, isn't it? Is there any other way to disambiguate our expression grammar?

Changing the Grammar to Impose Precedence

$$
\begin{aligned}
&<\text{expr}> ::= <\text{expr}> - <\text{expr}> \mid \\
&\qquad\qquad\quad <\text{term}> \\
&<\text{term}> ::= <\text{term}> * <\text{term}> \mid \\
&\qquad\qquad\quad <\text{factor}> \\
&<\text{factor}> ::= 0 \mid 1 \mid 2 \mid 3 \mid \ldots \mid 9
\end{aligned}
$$

Next Lecture

Expression grammars, precedence, associativity Top-down parsing, FIRST and FOLLOW sets, LL(1) grammars

Things to do:

- read Scott, Ch. 2.3 - 2.5 (skip 2.3.3 Bottom-up Parsing)