creating an abstract representation of program syntax
Today: Parsing

Source Code (Character stream)
```java
if (b == 0) { a = 1; }
```

Lexical Analysis

Token stream:
```java
if ( b == 0 ) { a = 0 ; }
```

Abstract Syntax Tree:
```
  If
    Eq
      b
    Assn
      0
      a
      1
    None
```

Parsing

Analysis & Transformation

Backend

Assembly Code
```
l1:
  cmpq %eax, $0
  jeq l2
  jmp l3
l2:
  ..."
Tokens:

10 + 5

Structure:

add 10 and 5

call a function $f$ with argument $x$

if statement with condition $b$ and body $a = 0$
Tokens:

10 + 5

Structure:

add 10 and 5

call a function \( f \) with argument \( x \)

if statement with condition \( b \) and body \( a = 0 \)
- Figure out what role each token is playing
- Catch most syntax errors ("valid pieces, bad arrangement")
- Understand what program the user wrote
- Turn it into a representation we can traverse and analyze

**Tokens:**

```
10 + 5
```

**Structure:**

```
add 10 and 5
```

```
call a function $f$ with argument $x$
```

```
if statement with condition $b$ and body that assigns 0 to $a$
```

```
if ( $b$ ) { $a = 0$ ; }
```
Parsing: Overview

• Input: stream of tokens (generated by lexer)
• Output: abstract syntax tree

• Strategy:
  – Parse the token stream to build a tree showing how the pieces relate
  – Forget the “concrete” syntax, remember the “abstract” syntax

• Will catch lots of malformed programs! Wrong number of operators, missing semicolons, unmatched parens; most “syntax errors” appear here
  – But no type errors, initialization, etc.: we still don’t know what anything means!
Describing Syntax

• How are we going to write these things down?

• Exercise: Describe the structure of an if statement, in terms of the tokens involved.

  “An if statement starts with the token IF, then an LPAREN, some tokens that make up a condition, and an RPAREN, then an LBRACE, some tokens that make up the body, and an RBRACE.”

if_stmt ::= IF LPAREN cond RPAREN LBRACE stmts RBRACE

• Note: regexps aren’t expressive enough for this, because they can’t really do recursion (example: can’t describe matching parentheses)
Questions

Top
CONTEXT-FREE GRAMMARS
Here is a specification of the language of balanced parens:

The definition is *recursive*: S mentions itself.

Idea: “derive” a string in the language by starting with S and rewriting according to the rules:

- Example:  
  \[ S \rightarrow (S)S \rightarrow (((S)S)S) \rightarrow (((\varepsilon)S)S) \rightarrow (((\varepsilon)\varepsilon)S) \rightarrow (((\varepsilon)\varepsilon)\varepsilon) = (()) \]

You can replace the “nonterminal” S by its definition anywhere.

A context-free grammar accepts a string when there is a derivation from the start symbol.
A Context-Free Grammar (CFG) consists of

- A set of terminals (e.g., a lexical token or ε)
- A set of nonterminals (e.g., S and other syntactic variables)
- A designated nonterminal called the start symbol
- A set of productions: \( LHS \rightarrow RHS \)
  - LHS is a nonterminal
  - RHS is a string of terminals and nonterminals

Example: The balanced parentheses language:

\[
S \rightarrow (S)S \\
S \rightarrow \varepsilon
\]

Exercise: How many terminals? How many nonterminals? Productions?
Another Example: Sum Grammar

- A grammar that accepts parenthesized sums of numbers:

  \[
  S \rightarrow E + S \quad | \quad E
  \]
  \[
  E \rightarrow \text{number} \quad | \quad ( S )
  \]

  e.g.: \((1 + 2 + (3 + 4)) + 5\)

- Note the vertical bar ‘|’ is shorthand for multiple productions:

  4 productions
  - \(S \rightarrow E + S\)
  - \(S \rightarrow E\)
  - \(E \rightarrow \text{number}\)
  - \(E \rightarrow (S)\)

  4 terminals: (, ), +, number
  2 nonterminals: \(S, E\)

  Start symbol: \(S\)
Derivations in CFGs

- Example: derive \((1 + 2 + (3 + 4)) + 5\)
- \(S \rightarrow E + S\)
  \(\rightarrow (S) + S\)
  \(\rightarrow (E + S) + S\)
  \(\rightarrow (1 + S) + S\)
  \(\rightarrow (1 + E + S) + S\)
  \(\rightarrow (1 + 2 + S) + S\)
  \(\rightarrow (1 + 2 + E) + S\)
  \(\rightarrow (1 + 2 + (S)) + S\)
  \(\rightarrow (1 + 2 + (E + S)) + S\)
  \(\rightarrow (1 + 2 + (3 + S)) + S\)
  \(\rightarrow (1 + 2 + (3 + E)) + S\)
  \(\rightarrow (1 + 2 + (3 + 4)) + S\)
  \(\rightarrow (1 + 2 + (3 + 4)) + E\)
  \(\rightarrow (1 + 2 + (3 + 4)) + 5\)

\[ S \rightarrow E + S \mid E \]
\[ E \rightarrow \text{number} \mid (S) \]

For arbitrary strings \(\alpha, \beta, \gamma\) and production rule \(A \rightarrow \beta\)
a single step of the derivation is:

\[ \alpha A\gamma \rightarrow \alpha \beta \gamma \]

(\textit{substitute} \(\beta\) for an occurrence of \(A\))

In general, there are many possible derivations for a given string

Note: Underline indicates symbol being expanded.
Loops and Termination

• Some care is needed when defining grammars

• Consider:

\[
S \rightarrow E \\
E \rightarrow S
\]

– This grammar has nonterminal definitions that are “nonproductive”. (i.e. they don’t mention any terminal symbols)
– There is no finite derivation starting from S, so the language is empty.

• Consider:

\[
S \rightarrow (S)
\]

– This grammar is productive, but again there is no finite derivation starting from S, so the language is empty

• When writing a large grammar, it’s easy to accidentally “chain” many nonterminals without a base case

• Upshot: be aware of “vacuously empty” CFG grammars.
– Every nonterminal should eventually rewrite to an alternative that contains only terminal symbols.
Questions

Top
debugging parser conflicts
disambiguating grammars
Getting Started with Yacc/Bison

• [https://www.gnu.org/software/bison/](https://www.gnu.org/software/bison/)
• [https://sourceforge.net/projects/gnuwin32/](https://sourceforge.net/projects/gnuwin32/) for Windows (but Cygwin or WSL works better)

• Run `yacc -d` or `bison -yd <grammar>.y` to get two files:
  – `y.tab.c` implements the parser
  – `y.tab.h` defines tokens and values for use in lexing – include in `.lex` file (replaces files like `tokens.h`)

• Not every grammar can be automatically parsed!
  – When run, reports number of shift/reduce and reduce/reduce conflicts
  – `yacc -dv` or `bison -ydv` also produces `y.output`, which describes the parser states and conflicts
Anatomy of a Yacc file

```c
#include "yacc.h"

int yylex(void);
%

union {
  int ival;
  char* sval;
}
%

token <ival> NUM

%%

exp:
  NUM { printf("number\n"); }
  | exp PLUS exp { printf("addition\n"); }
%

int main(){
  yyparse();
}
```
Yacc Actions

NUM
{ printf("number\n"); }

if we just want to know what’s in the program

NUM
{ $$ = $1; }

$$ is the return value for this production; $1 gets the value of the first symbol

exp PLUS exp
{ $$ = $1 + $3; }

access values of tokens and nonterminals by their position in the rule

• Later, we’ll build a representation of the program instead of running it right away!
Running the Lexer

• Running `yacc -d <filename>.y` generates `y.tab.c` and `y.tab.h`

• `y.tab.c` defines a function called `yyparse`, which parses a stream of tokens according to the grammar (using `yylex` to get the tokens)

• `y.tab.h` defines the tokens and their values, and should be used in the lexer instead of defining them there

• If parser has a main function, we can just compile and run `y.tab.c` together with the `lex.yy.c` from the lexer

• Otherwise, we can use the lexer and parser as a library, and call the generated `yyparse` function in other files (the rest of the compiler)

• Adding the `-v` argument (e.g. `yacc -dv <filename>.y`) also generates `y.output`, which can help with debugging (more on this later)
Questions
Ambiguity

- Consider this grammar:
  \[ S \rightarrow S - S \mid \text{number} \]

- How do we parse \( 1 - 2 - 3 \)?

  \[
  S \rightarrow S - S \\
  \rightarrow 1 - S \\
  \rightarrow \ldots
  \]

  \[
  S \rightarrow S - S \\
  \rightarrow S - 3 \\
  \rightarrow \ldots
  \]

  “This is an expression that computes 1 minus <an expression>”

  “This is an expression that computes <an expression> minus 3”
Ambiguity

• Consider this grammar:

\[
S \rightarrow S - S \mid \text{number}
\]

• How do we parse 1 – 2 – 3?

\[
\begin{align*}
S & \rightarrow S - S \\
& \rightarrow 1 - S \\
& \rightarrow 1 - S - S \\
& \rightarrow 1 - 2 - S \\
& \rightarrow 1 - 2 - 3
\end{align*}
\]

\[
\begin{align*}
S & \rightarrow S - S \\
& \rightarrow S - 3 \\
& \rightarrow S - S - 3 \\
& \rightarrow 1 - S - 3 \\
& \rightarrow 1 - 2 - 3
\end{align*}
\]

“This is an expression that computes 1 minus <2 minus 3>”

“This is an expression that computes <1 minus 2> minus 3”
• Consider this grammar: \( S \rightarrow S - S \mid \text{number} \)

• How do we parse \( 1 - 2 - 3 \)?

\[
\begin{align*}
S &\rightarrow S - S \\
&\rightarrow 1 - S \\
&\rightarrow 1 - S - S \\
&\rightarrow 1 - 2 - S \\
&\rightarrow 1 - 2 - 3
\end{align*}
\]

\[
\begin{align*}
S &\rightarrow S - S \\
&\rightarrow S - 3 \\
&\rightarrow S - S - 3 \\
&\rightarrow 1 - S - 3 \\
&\rightarrow 1 - 2 - 3
\end{align*}
\]

“This is an expression that does \( 1 - (2 - 3) \)”

“This is an expression that does \( (1 - 2) - 3 \)”
Associativity and Precedence

• Consider this grammar:

\[
S \rightarrow E - S \mid E \\
E \rightarrow \text{number} \mid (S)
\]

• This grammar makes ‘−’ right-associative

• If we want to generate 1 + 2 + 3:

\[
S \rightarrow E - S \\
\rightarrow 1 - S \\
\rightarrow 1 - E - S \\
\rightarrow 1 - 2 - S \\
\rightarrow 1 - 2 - E \\
\rightarrow 1 - 2 - 3
\]

but

\[
S \rightarrow E - S \\
\rightarrow E - E \\
\rightarrow E - 3 \\
\rightarrow \text{can’t make it!}
\]

• So 1 − (2 − 3) is the only possible parse

• Note that the grammar is right recursive!

• Exercise: How would you make ‘−’ left-associative?
Eliminating Ambiguity

• We can often eliminate ambiguity by layering the grammar (precedence) and allowing recursion only on one side (associativity).
• Higher-precedence operators go farther from the start symbol.
• Example:

\[ S \to S + S \mid S \ast S \mid (S) \mid \text{number} \]

• To disambiguate:
  – Decide (following math) to make ‘*’ higher precedence than ‘+’
  – Make ‘+’ left associative
  – Make ‘*’ right associative
• Note: \(S_2\) corresponds to ‘atomic’ expressions

\[
\begin{align*}
S_0 & \to S_0 + S_1 \mid S_1 \\
S_1 & \to S_2 \ast S_1 \mid S_2 \\
S_2 & \to \text{number} \mid (S_0)
\end{align*}
\]
Questions

Top
Precedence and Associativity Declarations

• Parser generators like yacc/bison often support precedence and associativity declarations
  – Resolve common cases of shift/reduce conflicts without refactoring

```%
%left PLUS
%left TIMES
```

• Tokens can be declared `left`, `right`, or `nonassoc`
• Tokens further down have higher precedence (bind tighter, get evaluated first)
• Precedence of a `rule` is based on the precedence of its `last` terminal:
  ```
  E → E + E
  1 + 2 . * 3
  E → E * E
  ```
• Can’t apply precedence to nonterminals
Example: Ambiguity in Real Languages

Consider this grammar:

\[
\begin{align*}
S &\rightarrow \text{if } (E) \ S \\
S &\rightarrow \text{if } (E) \ S \ \text{else } S \\
S &\rightarrow X = E \\
E &\rightarrow \ldots
\end{align*}
\]

Consider how to parse:

\[
\text{if } (E_1) \ \text{if } (E_2) \ S_1 \ \text{else } S_2
\]

This is known as the “dangling else” problem.

What should the “right” answer be?

How do we change the grammar?
How to Disambiguate if-then-else

• Want to rule out:

\[
\text{if (E}_1\text{)} \left\{ \begin{array}{l}
\text{if (E}_2\text{) S}_1 \\
\text{else S}_2
\end{array} \right.
\]

• Rule: An un-matched ‘if’ should not appear as the ‘then’ clause of a containing ‘if’.

\[
\begin{align*}
S &\rightarrow M \mid U \quad \text{ // } M = \text{“matched”}, \ U = \text{“unmatched”} \\
U &\rightarrow \text{if } (E) \ S \quad \text{ // Unmatched ‘if’} \\
U &\rightarrow \text{if } (E) \ M \ \text{ else } U \quad \text{ // Nested if is matched} \\
M &\rightarrow \text{if } (E) \ M \ \text{ else } M \quad \text{ // Matched ‘if’} \\
M &\rightarrow X = E \quad \text{ // Other statements}
\end{align*}
\]
Alternative: Use { }

• Ambiguity arises because the ‘then’ branch is not well bracketed:

if (E₁) { if (E₂) { S₁ } } else S₂  // unambiguous
if (E₁) { if (E₂) { S₁ } else S₂ }  // unambiguous

• So: could just require brackets
  – But requiring them for the else clause too leads to ugly code for chained if-statements:

  ```java
  if (c1) {
      ...
  } else {
      if (c2) {
          ...
      } else {
          if (c3) {
              ...
          } else {
              ...
          }
      }
  }
  ```

  So, compromise? Allow unbracketed else block only if the body is ‘if’:

  ```java
  if (c1) {
      ...
  } else if (c2) {
      ...
  } else if (c3) {
      ...
  } else {
      ...
  }
  ```

Benefits:
• Less ambiguous
• Easy to parse
• Enforces good style
Questions

Top
Expressions and Statements

• (demo)