creating an abstract representation of program syntax
Today: Parsing

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

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

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

Assembly Code
11:
cmpq %eax, $0
jeq 12
jmp 13
12:
...
{ 
    if (b == 0) a = b;
    while (a != 1) {
        print_int(a);
        a = a - 1;
    }
}
Syntactic Analysis (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

• Why abstract? Consider these three different concrete inputs:
  
  \[
  \begin{align*}
  a & + b \\
  (a &+ ( (b) )) \\
  ((a) &+ (b))
  \end{align*}
  \]

  Same abstract syntax tree

• 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!
Specifying Language Syntax

• First question: how to describe language syntax precisely and conveniently?

• Last time: we described tokens using regular expressions
  – Easy to implement, efficient DFA representation
  – Why not use regular expressions on tokens to specify programming language syntax?

• Limits of regular expressions:
  – DFAs have only finite # of states
  – So DFAs can’t “count”
  – For example, consider the language of all strings that contain balanced parentheses – easier than most programming languages, but not regular.

• We need more expressive power than DFAs!
CONTEXT-FREE GRAMMARS
Here is a specification of the language of balanced parens:

The definition is \textit{recursive} – \( S \) mentions itself.

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

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

You can replace the “\textit{nonterminal}” \( S \) by its definition anywhere.

A context-free grammar accepts a string iff 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: \( \text{LHS} \rightarrow \text{RHS} \)
  - LHS is a nonterminal
  - RHS is a *string* of terminals and nonterminals

Example: The balanced parentheses language:

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

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

A grammar that accepts parenthesized sums of numbers:

- $S \rightarrow E + S \mid E$
- $E \rightarrow \text{number} \mid (S)$

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

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

- 4 productions
- 2 nonterminals: $S, E$
- 4 terminals: (, ), +, number
- Start symbol: $S$
Derivations in CFGs

- Example: derive \((1 + 2 + (3 + 4)) + 5\)
  - \(S \rightarrow E + S\)
    - \((S) + S\)
    - \((E + S) + S\)
    - \((1 + S) + S\)
    - \((1 + E + S) + S\)
    - \((1 + 2 + S) + S\)
    - \((1 + 2 + E) + S\)
    - \((1 + 2 + (S)) + S\)
    - \((1 + 2 + (E + S)) + S\)
    - \((1 + 2 + (3 + S)) + S\)
    - \((1 + 2 + (3 + E)) + S\)
    - \((1 + 2 + (3 + 4)) + S\)
    - \((1 + 2 + (3 + 4)) + E\)
    - \((1 + 2 + (3 + 4)) + 5\)
  
- Production rule:
  \[ S \rightarrow E + S \mid E \]
  \[ E \rightarrow \text{number} \mid (S) \]
  
For arbitrary strings \(\alpha, \beta, \gamma\) and production rule \(A \rightarrow \beta\)

a single step of the derivation is:

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

(substitute \(\beta\) for an occurrence of \(A\))

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

Note: Underline indicates symbol being expanded.
Questions
From Derivations to Parse Trees

- Tree representation of the derivation
- Internal nodes are nonterminals
  - Children are parts of the production used on that nonterminal
- Leaves of the tree are terminals
  - In-order traversal yields the input sequence of tokens
- No information about the order of the derivation steps

(1 + 2 + (3 + 4)) + 5
From Parse Trees to Abstract Syntax

- **Parse tree**: “concrete syntax”
  - **Abstract syntax tree (AST)**: Hides, or *abstracts*, unneeded information
From Parse Trees to Abstract Syntax

- Internal nodes are *connective* terminals
- Leaves are *atomic* terminals
- Nonterminals don’t appear at all!
- Captures logical structure of programs

- **Abstract syntax tree** (AST):

```
                  +
                 +  5
                1  +
               2  +
              3  4
```

- Hides, or *abstracts*, unneeded information
Derivation Orders

• Productions of the grammar can be applied in any order.

• There are two standard orders:
  – *Leftmost derivation*: Find the left-most nonterminal and apply a production to it.
  – *Rightmost derivation*: Find the right-most nonterminal and apply a production there.

• Note that both strategies (and any other) yield the same parse tree!
  – Parse tree doesn’t contain the information about what order the productions were applied.
Example: Left- and rightmost derivations

- Leftmost derivation:
  \[ S \rightarrow E + S \]
  \[ \rightarrow (S) + S \]
  \[ \rightarrow (E + S) + S \]
  \[ \rightarrow (1 + S) + S \]
  \[ \rightarrow (1 + E + S) + S \]
  \[ \rightarrow (1 + 2 + S) + S \]
  \[ \rightarrow (1 + 2 + E) + S \]
  \[ \rightarrow (1 + 2 + (S)) + S \]
  \[ \rightarrow (1 + 2 + (E + S)) + S \]
  \[ \rightarrow (1 + 2 + (3 + S)) + S \]
  \[ \rightarrow (1 + 2 + (3 + E)) + S \]
  \[ \rightarrow (1 + 2 + (3 + 4)) + S \]
  \[ \rightarrow (1 + 2 + (3 + 4)) + E \]
  \[ \rightarrow (1 + 2 + (3 + 4)) + 5 \]

- Rightmost derivation:
  \[ S \rightarrow E + S \]
  \[ \rightarrow E + E \]
  \[ \rightarrow E + 5 \]
  \[ \rightarrow (S) + 5 \]
  \[ \rightarrow (E + S) + 5 \]
  \[ \rightarrow (E + E + S) + 5 \]
  \[ \rightarrow (E + E + E) + 5 \]
  \[ \rightarrow (E + E + (S)) + 5 \]
  \[ \rightarrow (E + E + (E + S)) + 5 \]
  \[ \rightarrow (E + E + (E + E)) + 5 \]
  \[ \rightarrow (E + E + (E + 4)) + 5 \]
  \[ \rightarrow (E + E + (3 + 4)) + 5 \]
  \[ \rightarrow (E + 2 + (3 + 4)) + 5 \]
  \[ \rightarrow (1 + 2 + (3 + 4)) + 5 \]
Loops and Termination

- Some care is needed when defining CFGs
- 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

- Easily generalize these examples to a “chain” of many nonterminals, which can be harder to find in a large grammar

- Upshot: be aware of “vacuously empty” CFG grammars.
  - Every nonterminal should eventually rewrite to an alternative that contains only terminal symbols.
Questions
Homework 1 Questions

Top
2.1

• Strings where the first $a$ precedes the first $b$:

$c^*(a(a|b|c)^*)$?

• Strings that don’t contain $baa$:

$b|c$

$(b|c|a(b|c))^*a$

$(a|c|b+a?c)^*(b+a?)^*$
associativity, ambiguity, and precedence

GRAMMARS FOR PROGRAMMING LANGUAGES
Consider the input: $1 + 2 + 3$

Leftmost derivation:

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

Rightmost derivation:

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

Add 2 and 3
Then, add 1 and the result

\[
\begin{align*}
\text{AST} & \quad + \\
1 & \quad + \\
2 & \quad 3
\end{align*}
\]
Associativity

• This grammar makes ‘+’ \textit{right-associative}...
• The abstract syntax tree is the same for both $1 + 2 + 3$ and $1 + (2 + 3)$
• Note that the grammar is \textit{right recursive}...

\[
\begin{align*}
S & \rightarrow E + S \mid E \\
E & \rightarrow \text{number} \mid (S)
\end{align*}
\]

• How would you make ‘+’ \textit{left} associative?
• What are the trees for “$1 + 2 + 3$”?
Ambiguity

• In a straightforward grammar, there may be multiple possible trees for a single string.

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

\[
\begin{array}{c}
\text{AST 1} \\
1 & 2 & - & 3
\end{array}
\]

\[
\begin{array}{c}
\text{AST 2} \\
1 & - & 2 & 3
\end{array}
\]

• Is \( 1 - 2 - 3 \) the same as \( (1 - 2) - 3 \) or \( 1 - (2 - 3) \)?
• Is \(-\) \textit{left-associative} or \textit{right-associative}?
In a straightforward grammar, there may be multiple possible trees for a single string.

```
S \rightarrow S + S \mid S * S \mid (S) \mid \text{number}
```

Is $1 + 2 * 3$ the same as $(1 + 2) * 3$ or $1 + (2 * 3)$?

Does + or * have higher precedence?
Eliminating Ambiguity

• We can often eliminate ambiguity by adding nonterminals and allowing recursion only on the left (or right).
• Higher-precedence operators go farther from the start symbol.
• Example:

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

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

\[
S_0 \longmapsto S_0 + S_1 \mid S_1 \\
S_1 \longmapsto S_2 \ast S_1 \mid S_2 \\
S_2 \longmapsto \text{number} \mid (S_0)
\]
Eliminating Ambiguity

\[
S_0 \rightarrow \\
S_0 + S_1 \\
S_0 + S_2 \ast S_1 \\
\ldots \rightarrow 1 + 2 \ast 3
\]

- \( 1 + 2 \ast 3 \)

\[
S_0 \rightarrow S_0 + S_1 \mid S_1 \\
S_1 \rightarrow S_2 \ast S_1 \mid S_2 \\
S_2 \rightarrow \text{number} \mid (S_0)
\]
Eliminating Ambiguity

$S_0 \rightarrow$
$S_1 \rightarrow$
$S_2 \ast S_1 \rightarrow$
$!!$

- $1 + 2 \ast 3$

\[
\begin{align*}
S_0 & \rightarrow S_0 + S_1 \mid S_1 \\
S_1 & \rightarrow S_2 \ast S_1 \mid S_2 \\
S_2 & \rightarrow \text{number} \mid (S_0)
\end{align*}
\]
Context Free Grammars: Summary

• Context-free grammars allow concise specifications of programming languages.
  – Can apply rules to convert token stream to parse tree/AST
  – Ambiguity can often be removed by encoding precedence and associativity in the grammar.

• Even with an unambiguous CFG, there may be more than one derivation
  – But all derivations correspond to the same abstract syntax tree.

• Next up: finding a derivation automatically (parsing!)
Questions

Top