CIS 341: COMPILERS

Lecture 10
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

Assn

None

b

0

a

1

Intermediate code:
l1:
  %cnd = icmp eq i64 %b, 0
  br i1 %cnd, label %l2, label %l3
l2:
  store i64* %a, 1
  br label %l3
l3:

Analysis & Transformation

Backend

Assembly Code

l1:
  cmpq %eax, $0
  jeq l2
  jmp l3
l2:
  ...
  ...
\{ 
    if (b == 0) a = b;
    while (a != 1) {
        print_int(a);
        a = a - 1;
    }
\}  

Source input

Abstract Syntax tree
Syntactic Analysis (Parsing): Overview

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

• Strategy:
  – Parse the token stream to traverse the “concrete” syntax
  – During traversal, build a tree representing the “abstract” syntax

• Why abstract? Consider these three different concrete inputs:
  \[ a + b \]
  \[ (a + ((b))) \]
  \[ (((a) + (b))) \]

  \[ \text{Same abstract syntax tree} \]

• Note: parsing doesn’t check many things:
  – Variable scoping, type agreement, initialization, …
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:
  – DFA’s have only finite # of states
  – So… DFA’s can’t “count”
  – For example, consider the language of all strings that contain balanced parentheses – easier than most programming languages, but not regular.

• So: we need more expressive power than DFA’s
CONTEXT FREE GRAMMARS
Context-free Grammars

• Here is a specification of the language of balanced parens:

\[
\begin{align*}
S & \rightarrow (S)S \\
S & \rightarrow \varepsilon
\end{align*}
\]

• 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: \( S \rightarrow (S)S \rightarrow ((S)S)S \rightarrow ((\varepsilon)S)S \rightarrow ((\varepsilon)S)\varepsilon \rightarrow ((\varepsilon)\varepsilon)\varepsilon = (()) \)

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

• A context-free grammar accepts a string iff there is a derivation from the start symbol

Note: Once again we have to take care to distinguish meta-language elements (e.g. “S” and “\(\rightarrow\)”) from object-language elements (e.g. “(”).”

* And, since we’re writing this description in English, we are careful distinguish the meta-meta-language (e.g. words) from the meta-language and object-language (e.g. symbols) by using quotes.
• A Context-free Grammar (CFG) consists of
  – A set of terminals (e.g., a token or $\varepsilon$)
  – 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:

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

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

• Note the vertical bar ‘\(|\)’ is shorthand for multiple productions:

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

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

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

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

For arbitrary strings $\alpha, \beta, \gamma$ and production rule $A \rightarrow \beta$

a single step of the derivation is:

$\alpha A \gamma \rightarrow \alpha \beta \gamma$

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

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

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

• Tree representation of the derivation

• Leaves of the tree are terminals
  – In-order traversal yields the input sequence of tokens

• Internal nodes: nonterminals

• No information about the order of the derivation steps

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

- **Parse tree**: “concrete syntax”
  - S
  - E + S
  - ( S ) E
  - E + S
  - E
  - 1
  - 2
  - 3
  - 4

- **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:
    
    \[
    \begin{align*}
    S &\rightarrow E \\
    E &\rightarrow S \\
    
    \end{align*}
    \]
    – 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.
Associativity, ambiguity, and precedence.
Consider the input: \( 1 + 2 + 3 \)

Leftmost derivation:
\[
\begin{align*}
S & \rightarrow E + S \\
& \rightarrow 1 + S \\
& \rightarrow 1 + E + S \\
& \rightarrow 1 + 2 + S \\
& \rightarrow 1 + 2 + E \\
& \rightarrow 1 + 2 + 3
\end{align*}
\]

Rightmost derivation:
\[
\begin{align*}
S & \rightarrow E + S \\
& \rightarrow E + E + S \\
& \rightarrow E + E + E \\
& \rightarrow E + E + 3 \\
& \rightarrow E + 2 + 3 \\
& \rightarrow 1 + 2 + 3
\end{align*}
\]
• This grammar makes ‘+’ right associative…
• The abstract syntax tree is the same for both 1 + 2 + 3 and 1 + (2 + 3)
• Note that the grammar is right recursive…

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

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

• Consider this grammar:

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

• Claim: it accepts the \textit{same} set of strings as the previous one.
• What’s the difference?
• Consider these \textit{two} leftmost derivations:
  
  – \[
  S \rightarrow S + S \rightarrow 1 + S \rightarrow 1 + S + S \rightarrow 1 + 2 + S \rightarrow 1 + 2 + 3
  \]
  
  – \[
  S \rightarrow S + S \rightarrow (S) + S \rightarrow 1 + S + S \rightarrow 1 + 2 + S \rightarrow 1 + 2 + 3
  \]

• One derivation gives left associativity, the other gives right associativity to ‘+’
  – Which is which?

\[
\begin{array}{c}
\text{AST 1} \\
+ \\
+ \\
\text{1} \quad \text{2}
\end{array}
\quad
\begin{array}{c}
\text{AST 2} \\
+ \\
+ \\
\text{2} \quad \text{3}
\end{array}
\]
Why do we care about ambiguity?

• The ‘+’ operation is associative, so it doesn’t matter which tree we pick. Mathematically, \( x + (y + z) = (x + y) + z \)
  – But, some operations aren’t associative. Examples?
  – Some operations are only left (or right) associative. Examples?

• Moreover, if there are multiple operations, ambiguity in the grammar leads to ambiguity in their precedence.

• Consider:

\[
S \leftrightarrow S + S \mid S \times S \mid (S) \mid \text{number}
\]

• Input: 1 + 2 * 3
  – One parse = \((1 + 2) * 3 = 9\)
  – The other = \(1 + (2 * 3) = 7\)

\[
\begin{align*}
\text{1} + 2 & \quad \times \quad \text{3} \quad \text{vs.} \\
\text{1} & \quad \text{2} & \quad \text{1} + 2 & \quad \times \\
& \quad \text{3} & \quad \text{2} & \quad \text{3}
\end{align*}
\]
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:

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

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

\[
\begin{align*}
S_0 & \rightarrow S_0 + S_1 \mid S_1 \\
S_1 & \rightarrow S_2 \times S_1 \mid S_2 \\
S_2 & \rightarrow \text{number} \mid (S_0)
\end{align*}
\]
CFGs Summary

• Context-free grammars allow concise specifications of programming languages.
  – An unambiguous CFG specifies how to parse: convert a token stream to a (parse tree)
  – 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
  – Though all derivations correspond to the same abstract syntax tree.

• Still to come: finding a derivation
  – But first: yacc
DEMO: BOOLEAN LOGIC

parser.mly, lexer.mll, range.ml, ast.ml, main.ml