Parsing

- Parser
  - Input: tokens
  - Output: abstract syntax tree, and a full understanding of the syntax of the language (but not the semantics)
- Author's interface:
  - TreeNode *syntaxTree = parse();
- A more OOP approach
  - Reference grammar on pg 492
  - We define a class called Program
    - It contains one instance variable – a declaration-list
  - The Parser returns a Program object
    - Root of AST
  - The Parser constructor takes a Scanner argument

Parsing

- Parser

```
public class CMinusParser implements Parser {

    private Scanner scan;

    public CMinusParser (String file) {
        scan = new Scanner(file);
    }

    public Program parse() {

    }
}
```

- Calling the Parser

```
String filename = “test”;
String sourceFile = fileName + “.c”;
Parser myParser = new CMinusParser (sourceFile);
Program myProgram = myParser.parse();
myProgram.printTree();
```

Parsing

- That's the big picture of the Parser – now we just need to define the guts of the parse() method
- As you might guess, this is a non-trivial task
  - Example:
    - The parser sees the keyword if
    - It then expects to see a (
      - If not, error
    - It then expects an Expression – something that resolves to a boolean
      - But this can be arbitrarily complex
    - Next it must find the closing)
    - Then it must find a statement
      - Again, can be arbitrarily complex
    - There may/may not be an else section
      - Wait till we discuss nested if statements!

Parsing

- The Parser is obviously complex
  - Note the recursive nature of the language definition
- As a result, it is by far the most studied part of compiler
  - Lots of different techniques we will examine
    - Chap 4 – top-down
    - Chap 5 – bottom-up

- To help deal with the complications, compiler writers have found that the use of Context-Free Grammars (CFGs) is very helpful to describe the syntax which the compiler is implementing
  - CGFs are somewhat similar to Regular Expressions, but are a bit more powerful for describing things like recursion

- So, what is a grammar (the common definition) ?

Parsing

- A grammar defines the legal way of putting words together to make sentences and paragraphs

- In CS, a context-free grammar does about the same thing
  - It takes tokens, and defines the legal way for stringing them together to make syntactically correct code
  - Legal strings of tokens also called sentences
  - Context-free means I can apply the language structure anywhere without concern to the context – if I am parsing an if statement, I can put any legal statement in the then and else sections – CFGs can't describe declare before use
- Production rules for grammars were first developed formally to describe the Algol60 language
  - Done by John Backus and adapted by Peter Naur
  - Grammar rules written like: expr -> expr op expr are commonly said to be in Backus-Naur form, or BNF

Context-Free Grammars

- A context-free grammar consists of the following
  1. A set T of terminals
     - The tokens of the grammar
  2. A set N of non-terminals
     - A structure which captures some language feature
     - Examples: program, statement, expression, function declaration
  3. A set P of productions or grammar rules
     - Of the form A -> α, where A is a non-terminal on left side, and α represents a string of either terminals or non-terminals on the right side of ->
  4. A start symbol S from the set N
     - The base non-terminal symbol
     - Typically is something like Program

Context-Free Grammars

- The format for expressing CFGs varies widely between texts
  - Probably the most common is program -> declaration-list
  - Others common forms
    - program = declaration-list
    - program : declaration-list
    - program :: declaration-list
    - Program ::= declaration-list
- Compare a CFG to a Regular Expression
  - Both have choice and concatenation
  - REs use closure
  - CFGs use recursion (more powerful)

```
number = digit digit*
digit = [0..9]
```

```
term -> term mulop factor | factor
mulop -> * | /
factor -> (expr) | var | call | NUM
```

Context-Free Grammars

- The author goes into great detail expressing his notation for differentiating between terminal and non-terminal
  - I think the context should make it fairly obvious

- Starting with the start symbol (*program*), we can expand one of the non-terminals of the right-hand side of the production using some legal production
  - If we continue to do this until we just have terminal symbols on the right-hand side, we have found a legal sentence of the language
  - We call the creation of this legal sentence a derivation
    - A derivation consists of a sequence of productions
    - Each step typically shown using => symbol

Context-Free Grammars

- A possible derivation for if (a == b) { c = d; }
  - stmt => if_stmt

    => if (expr) stmt

    => if (expr binop expr) stmt

    => if (IDENT binop expr) stmt

    => if (IDENT == expr) stmt

    => if (IDENT == IDENT) stmt

    => if (IDENT == IDENT) cmpd_stmt

    => if (IDENT == IDENT) { stmt }

    => if (IDENT == IDENT) { expr_stmt }

    => if (IDENT == IDENT) { expr = expr; }

    => if (IDENT == IDENT) {IDENT = expr; }

    => if (IDENT == IDENT) {IDENT = IDENT; }

Context-Free Grammars

- If we start from the start symbol, we can come up with many (usually infinite) possible sentences
  - We call this the language defined by the grammar
    - Symbolically: L(G) for language defined by grammar G

- Example: What is the language defined by the grammar given by : E -> (E) | a
  - Note: we couldn't have described this with Regular Expressions. Why?

- Example: What is the language defined by the grammar given by : E -> (E)

Context-Free Grammars

- Example: What is the language defined by the grammar given by : E -> E + a | a

- Example: Consider the following grammar

  expr -> expr op expr

  expr -> (expr)

  expr -> - expr

  expr -> IDENT

  op -> + | - | * | /

  - What are the terminal/non-terminals?
  - What are some possible derivations?
  - What is a derivation for a + b * (d + e)?
    - Is there another derivation?

Context-Free Grammars

- If we can apply a set of productions to get from one form to another, frequently we use the symbol =>*
  - stmt => if_stmt

    => if (expr) stmt

    => if (expr binop expr) stmt

    => if (IDENT binop expr) stmt
  - stmt =>* if (IDENT binop expr) stmt

    Means I can get from stmt to the right-hand side form through a series of 1 or more productions

- We said that CFGs are more powerful than Regular Expressions
  - What about closure?

Context-Free Grammars

- Example: A -> Aa | a
  - What language does this describe?
  - Is this closure?

- To fully do closure, we need to be able to generate an empty string
  - To do this, we allow productions which generate empty strings
  - Uses the symbol ε, e.g., A -> ε
- Now we can define a production A -> Aa | ε
  - What language does this describe?
  - Is this closure?

Context-Free Grammars

- We saw that A -> Aa | ε essentially is same as a*
- What about the production: A -> aA | ε ?
  - What language does this produce?

- Are the two forms exactly the same?
- Does it matter which form you choose?
  - Well, it turns out that it is critical
  - Note that the form A -> Aa grows at the front
    - The first production creates the last “a” in the sequence
    - Subsequent “a”s are pre-pended
    - We call this form left recursion – the recursive non-terminal is 1st symbol on right side of production
  - The other form causes the “a”s to grow onto the end of string
    - We call this form right recursion

Context-Free Grammars

- It isn't obvious now, but building your grammar using either left or right recursion becomes critical for the different kinds of parsers

- Example: what does the following grammar represent?

  A -> (A) A | ε

- Example: inside a compound statement { }, you can put a sequence of zero of more statements, with a semicolon after each
  - How would you represent this using grammar productions?
    - stmt_sequence -> stmt ; stmt_sequence | ε
    - stmt -> s
  - Produces s;s;s;

Context-Free Grammars

- Note that the previous example produced a string of s which always had a ; afterward
  - For some languages, the last statement in a block shouldn't have a ; at the end
  - To do this, you have to play some tricks
    - stmt_sequence -> ne_stmt_sequence | ε
    - ne_stmt_sequence -> stmt ; ne_stmt_sequence | stmt
    - stmt -> s

- A new definition – leftmost derivation
  - Always expands the first (leftmost) non-terminal on right side of the production
  - A rightmost derivation always expands last non-terminal

Context-Free Grammars

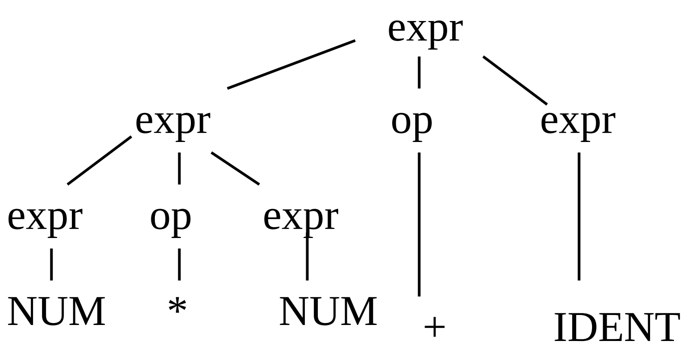
- How do derivations and parse trees relate?
  - A parse tree can be built, top-down, by looking at the productions executed
  - Consider the grammar below and a leftmost derivation
    - expr => expr op expr

      => expr op expr op expr

      => NUM op expr op expr

      => NUM * expr op expr ….

```
expr -> expr op expr
expr -> ( expr )
expr -> - expr
expr -> IDENT | NUM
op -> + | - | * | /
```



How would tree differ with a rightmost derivation?

Context-Free Grammars

- You can see it is definitely important whether you do a leftmost or rightmost derivation
  - Top-down parsers are based on a leftmost derivation
  - Bottom-up parser implementations depend on doing reductions which correspond to a rightmost derivation

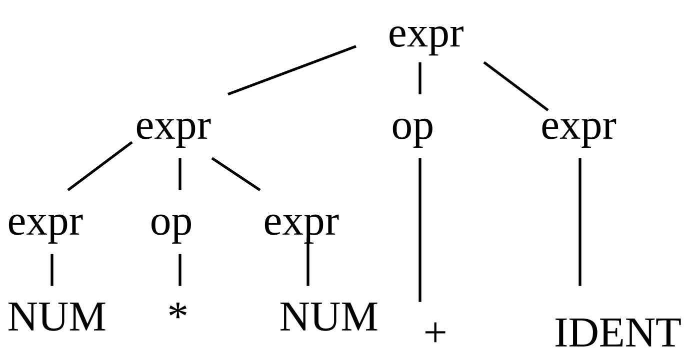
- A parse tree maintains a one-to-one relationship with the grammar productions it represents
  - However, it maintains some extra baggage which isn't really needed to fully represent the HLL language
    - Impacts efficiency
  - A more succinct representation, which still retains the full meaning of the code, is the Abstract Syntax Tree (AST)
    - Typically, parsers build an AST directly, and never generate the actual Parse Tree representation

Context-Free Grammars

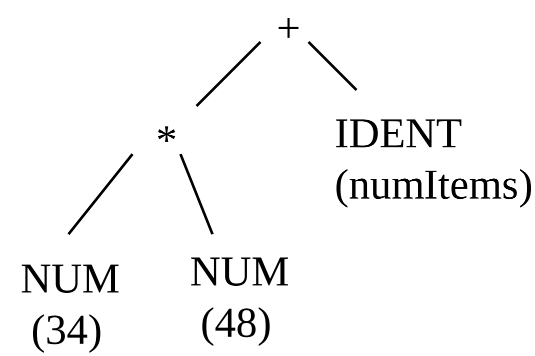
- The following trees both represent a piece of code like:

  34 * 48 + numItems

  - The AST is quite a bit cleaner
  - We will use the AST, but need to understand how it represents the grammar productions
  - Note: the AST corresponds one-to-one with our internal data structures



Parse Tree



Abstract Syntax Tree

Ambiguity

- Context-Free Grammars work very well for expressing the syntax of a language
  - However, they aren't perfect, and certain language features give them difficulties
- Typically, for a particular sentence, there should only be one AST to describe it
  - The classic example is: a = b + c * d
    - An AST which does the add before the multiply is not so good
- If a grammar can result in more than one AST for a given sentence, the grammar is said to be ambiguous
  - Another way of saying this: there is more than one leftmost or more than one rightmost derivation for a given sentence

Ambiguity

- Consider b + c * d or ident + ident * ident
- Two leftmost derivations – what do the ASTs look like?
- Is this an ambiguity?

```
Expr => expr op expr
     => expr op expr op expr
     => IDENT op expr op expr
     => IDENT + expr op expr
     => IDENT + IDENT op expr
     => IDENT + IDENT * expr
     => IDENT + IDENT * IDENT
```

```
Expr => expr op expr
     => IDENT op expr
     => IDENT + expr
     => IDENT + expr  op expr
     => IDENT + IDENT op expr
     => IDENT + IDENT * expr
     => IDENT + IDENT * IDENT
```

```
expr -> expr op expr
expr -> ( expr )
expr -> - expr
expr -> IDENT | NUM
op -> + | - | * | /
```

Ambiguity

- Ambiguity – would not be a good thing
  - What can be done about it? Two basic approaches:
    1. Live with it (work around it)
    2. Change grammar to remove ambiguity

- Living with ambiguity
  - Treat each occurrence as a special case in your parser
    - If precedence of operators causes difficulty, have your parser look for this situation and handle it
  - Rules we put in the parser are known as disambiguating rules
  - Advantage of this approach is that we don't have to change the grammar, which typically complicates it
  - Disadvantage is it complicates parser
    - Special cases aren't usually a good idea … why?

Ambiguity

- Consider our previous example: a = b + c * d
  - When it sees b + c, does it make a reduction or go look at the next token?
    - We can write a special case in the parser to look ahead at next character and make the decision based on that
- By defining that * and / have higher precedence than + and -, we can fix this problem
  - However, we still have a problem with what to do with a string of +
- Consider a = b + c + d + e and also a = b = c = d = e
  - Which operation happens first in 1st case?
  - Which operation happens first in 2nd case?
  - We need a rule for this also

Ambiguity

- In addition to defining precedence, we also define associativity
  - We say that + and – are left associative
    - A series of adds is done left to right
  - We say that = is right associative
- Obviously, we will have to do a special case in parser to handle associativity also

- The 2nd option for handling ambiguity is to change the grammar to remove it
  - Grammar would now fully capture the syntax
  - But the grammar might get more complicated

  - However, this is the preferred approach

Ambiguity

- Back to: b + c * d or ident + ident * ident
  - We were using the name op to represent all 4 operations
  - Let's divide this into 2 different names
  - We also add 2 new non-terminals: term and factor
  - If you have an addop and mulop side by side, the addop production must occur first in derivation
    - Which means the mulop occurs lower in tree, and is computed first!

```
expr -> expr op expr
expr -> ( expr )
expr -> IDENT | NUM
op -> + | - | * | /
```

Old way

```
expr -> expr addop expr | term
term -> term mulop term | factor
factor -> ( expr ) | IDENT | NUM
addop -> + | -
mulop -> * | /
```

New way

Ambiguity

- Success … well, not quite
  - What problem did we not solve?
- We handled the precedence problem, but not the associativity problem
  - The grammar is still ambiguous
  - A string of adds could result in different ASTs
  - By using expr -> expr addop expr we can expand expr on either side of addop
    - We fix by changing this line

expr -> expr addop expr | term term -> term mulop term | factor factor -> (expr) | IDENT | NUM addop -> + | - mulop -> * | /	expr -> expr addop term | term term -> term mulop factor | factor factor -> (expr) | IDENT | NUM addop -> + | - mulop -> * | /

Ambiguity

- Now, addition and multiplication are left associative, because the recursion has to be left recursion
- What about a string of = which need to be right associative?
  - How would we change assign -> assign = assign | expr ?

- That wasn't too bad – let's look at another classic language problem: the dangling else
  - How do the code sequences below differ?

```
if (a == b)
     if ( c == d)
          e = 1;
     else
          e = 0;
```

```
if (a == b)
     if ( c == d)
          e = 1;
else
     e = 0;
```

Ambiguity

- Remember, the compiler ignores your whitespace
- The rule the semantics of the language sets is that the else associates with the closest if statement
- Consider the (simplified) grammar below:
  - The sentence: if (0) if (1) other else other is ambiguous
  - Two different parse trees could be created

```
stmt -> if_stmt | other
if_stmt -> if  ( expr ) stmt  | if  ( expr ) stmt else stmt
expr -> 0 | 1
```

Ambiguity

- The ambiguity can be removed by re-writing the grammar
  - A bit more complicated than operator precedence
  - Differentiates matched and unmatched ifs

```
stmt -> if_stmt | other
if_stmt -> if  ( expr ) stmt  | if  ( expr ) stmt else stmt
expr -> 0 | 1
```

Old way

```
stmt -> matched_stmt | unmatched_stmt
matched_stmt -> if  ( expr ) matched_stmt else matched_stmt   | other
unmatched_stmt -> if  ( expr ) stmt
                  | if  ( expr ) matched_stmt else unmatched_stmt
expr -> 0 | 1
```

New way

Ambiguity

- So, why does this work?
  - Note that anywhere an else occurs, it must be proceeded by a matched_stmt
  - You cannot expand an unmatched statement (an if without an else) just before an else
  - else statements get associated correctly as soon as production is done

```
stmt -> matched_stmt | unmatched_stmt
matched_stmt -> if  ( expr ) matched_stmt else matched_stmt   | other
unmatched_stmt -> if  ( expr ) stmt
                              | if  ( expr ) matched_stmt else unmatched_stmt
expr -> 0 | 1
```

Context-Free Grammars

- It is better to correct your grammar rather than live with ambiguity
  - However, sometimes it is impossible to express a language construct with a CFG
  - CFGs are more powerful than Regular Expressions, but not perfect
- If a language can be generated by a CFG, we call it a context-free language
- Unfortunately, most programming languages are not context-free languages
  - They contain elements which cannot be expressed by a CFG

Context-Free Grammars

- Example: consider L = {wxw | w, x are (a|b)*}
  - In other words, two instances of the same string, separated by another string
  - It has been proven that this language is not context-free, and no grammar can describe this language
  - This example generalizes the requirement that an identifier has to be declared before being used
    - No context-free grammar can be defined to test this language requirement
  - Instead, we must check this on a subsequent pass through the code after parsing

Context-Free Grammars

- Example: Checking that the number of formal parameters to a function matches the number of parameters passed each time the function is invoked
  - Modeled as $a^m b^n c^m d^n$, where a, b are 2 function declarations, and c and d are call points for a and b
  - Again, this is not a context-free language, and cannot be expressed by a CFG
  - Checking done during subsequent pass

- BTW, this is where the term context-free comes from
  - CFGs imply that we can substitute any form of the RHS anywhere we want, without regard to the context
  - These previous 2 examples show this isn't always allowed in programming languages
- But, in general, CFGs can capture the vast majority of the syntax of typical HLLs

EBNF

- One last topic before we go back to implementation issues
  - Extended BNF
- Recall that Regular Expressions had a bunch of convenient extensions: a? a+ [0-9]
- The same idea is frequently used for BNF
  - It will be very helpful to you for your top-down parser

- Left/right recursion - uses { } to indicate repetition
  - stmt_sequence -> stmt ; stmt_sequence | stmt
  - (or) stmt_sequence -> { stmt ; } stmt
  - Left recursive form: stmt_sequence -> stmt { ; stmt }

  - Recall: expr -> expr addop term | term
  - Becomes: expr -> term { addop term }

EBNF

- Optional constructs
  - Shown inside [ ]
  - Example: else part of an if statement
    - if_stmt -> if (expr) stmt [else stmt]
  - Also useful for writing a statement in right-recursive form
    - expr -> term addop expr | term   becomes
    - expr -> term [addop expr]
  - Later we will look at why a left-recursive or right-recursive form might be preferable

Implementation of ASTs

- We said earlier that the parser actually implements an AST, not a Parse Tree
  - Let's look at how we might do this
- The author's approach (the C approach) is to define a generic TreeNode class which can act as any node in the AST
  - Most nodes should have left and right children
    - But what about an if-else – it has an expr and 2 statements as children
  - Also, we can do nothing better than to have the children pointers be pointers to Objects, since different nodes would have different things hanging on them
- A much better (and cooler) approach is to use an OO approach
  - Custom classes for each type of node

Implementation of ASTs

- Consider the if_stmt
  - It is of class IfStatement
  - An if_stmt can have 3 children: an expr and 2 stmts
    - So, our IfStatement class has instance variables:
      - expr, stmt1, and stmt2
  - But there are lots of types of expressions and statements
  - We don't know which type will actually be hooked up there
    - How do we handle something like this in OOP?

- We create abstract classes Expression and Statement
  - IfStatement extends Statement, as does WhileStatement, CompoundStatement, ReturnStatement, etc.
  - Make stmt1 and stmt2 of type Statement, and can hang anything there I want

Implementation of ASTs

- The IfStatement class

```
public class IfStatement extends Statement {

    Expression expr;
    Statement thenStmt;
    Statement elseStmt;

    public IfStatement (Expression express, Statement stmt) {
        this (express, stmt, null);
    }

    public IfStatement (Expression express, Statement stmt1, Statement stmt2) {
        expr = express;
        thenStmt = stmt1;
        elseStmt = stmt2;
    }
}
```

Implementation of ASTs

- Think about the statement sequence
  - EBNF might be stmt_seq -> stmt ; [stmt_seq]
  - If implemented directly from the EBNF, would have two instance vars: a Statement and a StatementSeq
    - An alternate, and more efficient form, is to use a linked structure like we did in the Heap project during Data Structures
      - Each stmt has a nextSibling pointer
      - Each time we add a new statement to a statement sequence, we just add it to the list
      - This makes walking the AST simpler and easier
  - How do we implement?
    - The abstract Statement class contains a nextSibling ref
    - All other statements inherit this from Statement
  - Or better yet … ArrayList<Statement>

Context-Free Grammars

- So, that’s it for Context-Free Grammars, and the AST
- With this background, we are ready to attack the Parser