Two Steps in the Algorithm to Parse Arbitrary Context-Free Grammars

1) Transform grammar to normal form
   called Chomsky Normal Form
   TODAY

2) Parse input using transformed grammar
   dynamic programming algorithm
   SEEN LAST WEEK

Conversion to Chomsky Normal Form (CNF)

Steps: (not in the optimal order)

- remove unproductive symbols
- remove unreachable symbols
- remove epsilons (no non-start nullable symbols)
- remove single non-terminal productions (unit productions)   X::=Y
- reduce arity of every production to less than two
- make terminals occur alone on right-hand side

1) Unproductive non-terminals

What is funny about this grammar:

```
stmt ::=  identifier := identifier
        | while (expr) stmt
        | if (expr) stmt else stmt
expr ::= term + term | term – term
term ::= factor * factor
factor ::= ( expr )
```

There is no derivation of a sequence of tokens from expr

In every step will have at least one expr, term, or factor

If it cannot derive sequence of tokens we call it *unproductive*

1) Unproductive non-terminals

Productive symbols are obtained using these two rules (what remains is unproductive)

- Terminals are productive
- If $X ::= s_1\ s_2 \ldots s_n$ is a rule and each s_i is productive then X is productive

Delete unproductive symbols.

The language recognized by the grammar will not change

2) Unreachable non-terminals

What is funny about this grammar with start symbol 'program'

program ::= stmt | stmt program
stmt ::= assignment | whileStmt
assignment ::= expr = expr
ifStmt ::= if (expr) stmt else stmt
whileStmt ::= while (expr) stmt
expr ::= identifier

No way to reach symbol 'ifStmt' from 'program'

Can we formulate rules for reachable symbols ?

2) Unreachable non-terminals

Reachable terminals are obtained using the following rules (the rest are unreachable)

- starting non-terminal is reachable (program)
- If X::= s_1 s_2 … s_n is rule and X is reachable then every non-terminal in s_1 s_2 … s_n is reachable

Delete unreachable nonterminals and their productions

3) Removing Empty Strings

Ensure only top-level symbol can be nullable

program ::= stmtSeq
stmtSeq ::= stmt | stmt ; stmtSeq
stmt ::= “” | assignment | whileStmt | blockStmt
blockStmt ::= { stmtSeq }
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
expr ::= identifier

How to do it in this example?

3) Removing Empty Strings - Result

program ::= “” | stmtSeq
stmtSeq ::= stmt | stmt ; stmtSeq |
| ; stmtSeq | stmt ; | ;
stmt ::= assignment | whileStmt | blockStmt
blockStmt ::= { stmtSeq } | { }
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
whileStmt ::= while (expr)
expr ::= identifier

3) Removing Empty Strings - Algorithm

$O(2^n)$

3) Removing Empty Strings

- Since stmtSeq is nullable, the rule

  blockStmt ::= { stmtSeq }

  gives

  blockStmt ::= { stmtSeq } | { }

- Since stmtSeq and stmt are nullable, the rule

  stmtSeq ::= stmt | stmt ; stmtSeq

  gives

  stmtSeq ::= stmt | stmt ; stmtSeq
  | ; stmtSeq | stmt ; | ;

4) Eliminating unit productions

- Single production is of the form

  X ::=Y

where X,Y are non-terminals

```
program ::= stmtSeq
stmtSeq ::= stmt
          | stmt ; stmtSeq
stmt ::= assignment | whileStmt
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
```

4) Unit Production Elimination Algorithm

- If there is a unit production

  X ::=Y put an edge (X,Y) into graph

- If there is a path from X to Z in the graph, and there is rule Z ::= s_1 s_2 … s_n then add rule

$$X ::= s_1 \; s_2 \; \ldots \; s_n$$

At the end, remove all unit productions.

4) Eliminate unit productions - Result

program ::= expr = expr | while (expr) stmt
                | stmt ; stmtSeq
stmtSeq ::= expr = expr | while (expr) stmt
                | stmt ; stmtSeq
stmt ::= expr = expr | while (expr) stmt
assignment ::= expr = expr
whileStmt ::= while (expr) stmt

5) Reducing Arity: No more than 2 symbols on RHS

stmt ::= while (expr) stmt

becomes

$stmt$ ::= while $stmt_1$

$stmt_1$ ::= ($stmt_2$

$stmt_2$ ::= expr $stmt_3$

$stmt_3$ ::=) stmt

6) A non-terminal for each terminal

$stmt ::= while\ (expr)\ stmt$

becomes

$stmt ::= N_{while}\ stmt_1$

$stmt_1 ::= N_{(}\ stmt_2$

$stmt_2 ::= expr\ stmt_3$

$stmt_3 ::= N_{)}\ stmt$

$N_{while} ::= while$

$N_{(} ::= ($

$N_{)} ::=)$

Order of steps in conversion to CNF

1. remove unproductive symbols (optional)
2. remove unreachable symbols (optional)
3. make terminals occur alone on right-hand side
4. Reduce arity of every production to <= 2
5. remove epsilons
6. remove unit productions X::=Y
7. unproductive symbols
8. unreachable symbols

– What if we swap the steps 4 and 5 ?
  - Potentially exponential blow-up in the # of productions

Ordering of Unreachable / Unproductive symbols

First Unreachable then Unproductive

S := B C | “”
C := D
D := a
R := r

S := B C | “”
C := D
D := a

S := “”
C := D
D := a

First Unproductive then Unreachable

S := B C | “”
C := D
D := C
R := r

S := “”
C := D
D := a
R := r

S := “”

Alternative to Chomsky form

We need not go all the way to Chomsky form

it is possible to directly parse arbitrary grammar

Key steps: (not in the optimal order)

- reduce arity of every production to less than two (otherwise, worse than cubic in string input size)

Can be less efficient in grammar size, but still works

More algorithms for arbitrary grammars are variations:

Earley's parsing algorithm (Earley, CACM 1970)

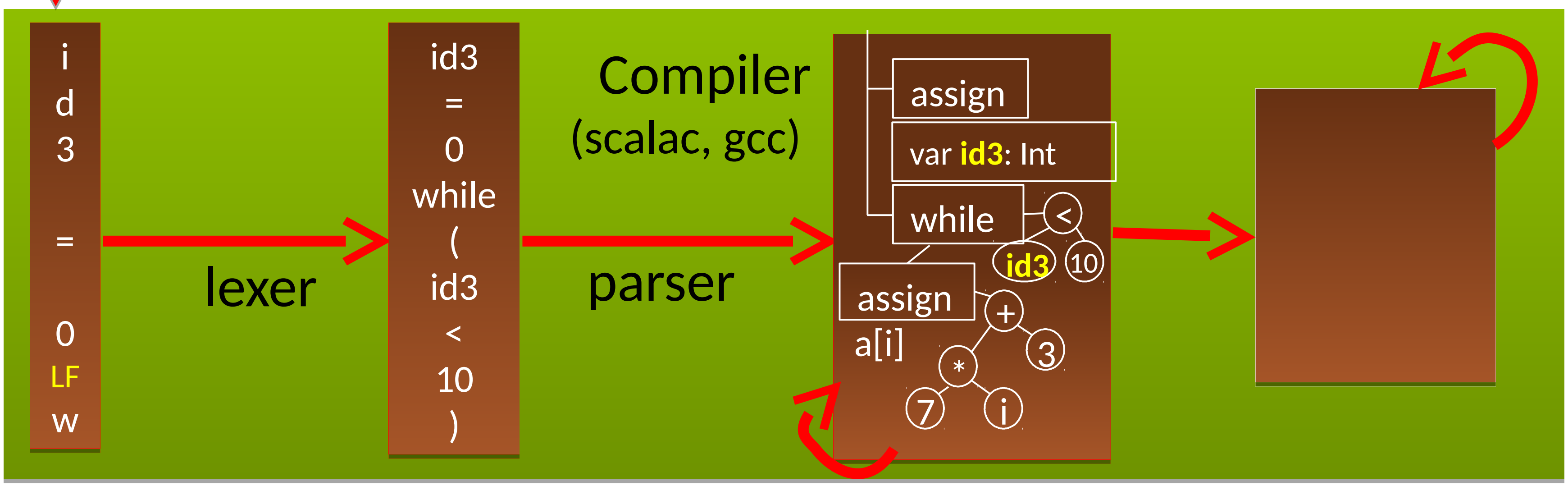
GLR parsing algorithm (Lang, ICALP 1974, Deterministic Techniques for Efficient Non-Deterministic Parsers)

GLL algorithm

```
Id3 = 0
while (id3 < 10) {
  println("",id3);
  id3 = id3 + 1 }
```

source code: sequence of characters

after each analysis the compiler has a better "understanding" of the input program; can report more subtle errors



characters

words (tokens)

trees

Name Analysis:

making sense of trees; converting them into **graphs**: connect identifier **uses** and **declarations**

Reporting Errors

Errors Detected So Far

- File input: file does not exist
- Lexer: unknown token, string not closed before end of file, ...
- Parser: syntax error - unexpected token, cannot parse given non-terminal
- Name analyzer: unknown identifier
- Type analyzer:
  applying function to arguments of wrong type
- Data-flow analyzer:
  variable read before written, division by zero

Name Analysis Problems Reported: 1

- a class is defined more than once:

  class A { ...} class B { ... } class A { ... }
- a variable is defined more than once:

  int x; int y; int x;
- a class member is overriden without **override** keyword:

  class A { int x; ... } class B extends A { int x; ... }
- a method is **overloaded** (forbidden in Tool):

  class A { def f(B x) {} def f(C x) {} ... }
- a method argument is shadowed by a local variable declaration (forbidden in Java, Tool):

  def (x:Int) { var x : Int; ...}
- two method arguments have the same name:

  def (x:Int,y:Int,x:Int) { ... }

Name Analysis Problems Reported: 2

- a class name is used as a symbol (as parent class or type, for instance) but is not declared:

  class A extends Objekt {}

- an identifier is used as a variable but is not declared:

  def(amount:Int) { total = total + ammount }

- the inheritance graph has a cycle:

  class A extends B {}

  class B extends C {}

  class C extends A

To make it efficient and clean to check for such errors, we associate mapping from each identifier to the symbol that the identifier represents.

- We use Map data structures to maintain this mapping
- The rules that specify how declarations are used to construct such maps are given by ***scoping* rules of the programming language.**

Storing and Using Tree Positions

Showing Good Errors with Syntax Trees

Suppose we have undeclared variable ‘i’ in a program of 100K lines

Which error message would you prefer to see from the compiler?

- An ocurrence of variable ‘i’ not declared (which variable? where?)
- An ocurrence of variable ‘i’ in procedure P not declared
- Variable ‘i’ undeclared at line 514, position 12 (and IDE points you there)ϟ

How to emit this error message if we only have a syntax trees?

- Abstract syntax tree nodes store positions within file
- For identifier nodes: allows reporting variable uses
  - Variable 'i' in line 11, column 5 undeclared
- For other nodes, supports useful for type errors, e.g. could report for (x + y) * (!ok)
  - Type error in line 13,
  - expression in line 13, column 11-15, has type **Bool**, expected **Int** instead

Showing Good Errors with Syntax Trees

Constructing trees with positions:

- Lexer records positions for tokens
- Each subtree in AST corresponds to some parse tree, so it has first and last token
- Get positions from those tokens
- Save these positions in the constructed tree

What is important is to save information for leaves

- information for other nodes can often be approximated using information in the leaves

Continuing Name Analysis:

Scope of Identifiers

Example: find program result, symbols, scopes

```
class Example {
  boolean x;
  int y;
  int z;
  int compute(int x, int y) {
    int z = 3;
    return x + y + z;
  }
  public void main() {
    int res;
    x = true;
    y = 10;
    z = 17;
    res = compute(z, z+1);
    System.out.println(res);
  }
}
```

Scope of a variable = part of the program where it is visible

Draw an arrow from occurrence of each identifier to the point of its declaration.

For each declaration of identifier, identify where the identifier can be referred to (its scope).

Name analysis:

- computes those arrows
  - = maps, partial functions (math)
  - = environments (PL theory)
  - = symbol table (implementation)
- report some simple semantic errors

We usually introduce **symbols** for things denoted by identifiers.
Symbol tables map identifiers to symbols.

Usual **static** scoping: What is the result?

```
class World {
  int sum;
  int value;
  void add() {
     sum = sum + value;
     value = 0;
  }
  void main() {
      sum = 0;
      value = 10;
      add();
      if (sum % 3 == 1) {
         int value;
         value = 1;
         add();
         print("inner value = ", value);  1
         print("sum = ", sum);  10
      }
      print("outer value = ", value);  0
  }
}
```

Identifier refers to the symbol that was declared “closest” to the place **in program structure** (thus "static").

We will assume static scoping unless otherwise specified.

Renaming Statically Scoped Program

```
class World {
  int sum;
  int value;
  void add(int foo) {
     sum = sum + value;
     value = 0;
  }
  void main() {
      sum = 0;
      value = 10;
      add();
      if (sum % 3 == 1) {
         int value1;
         value1 = 1;
         add(); // cannot change value1
         print("inner value = ", value1); 1
         print("sum = ", sum);   10
      }
      print("outer value = ", value);   0
   }
}
```

Identifier refers to the symbol that was declared “closest” to the place **in program structure** (thus "static").

We will assume static scoping unless otherwise specified.

Property of static scoping:
Given the entire program, we can **rename variables** to avoid any shadowing (**make** all vars **unique**!)

Dynamic scoping: What is the result?

```
class World {
  int sum;
  int value;
  void add() {
     sum = sum + value;
     value = 0;
  }
  void main() {
      sum = 0;
      value = 10;
      add();
      if (sum % 3 == 1) {
         int value;
         value = 1;
         add();
         print("inner value = ", value);  0
         print("sum = ", sum);  11
      }
      print("outer value = ", value);  0
  }
}
```

Symbol refers to the variable that was most **recently declared within program execution.**

Views variable declarations as executable statements that establish which symbol is considered to be the 'current one'. (Used in old LISP interpreters.)

Translation to normal code: access through a dynamic environment.

Dynamic scoping translated using global map, working like stack

```
class World {
  int sum;
  int value;
  void add() {
    sum = sum + value;
    value = 0;
  }
  void main() {
     sum = 0;
     value = 10;
     add();
     if (sum % 3 == 1) {
       int value;
       value = 1;
       add();
       print("inner value = ", value);   0
       print("sum = ", sum);   11
     }
     print("outer value = ", value);   0
   }
}
```

```
class World {
  pushNewDeclaration('sum);
  pushNewDeclaration('value);
  void add(int foo) {
    update('sum, lookup('sum) + lookup('value));
    update('value, 0);
  }
  void main() {
     update('sum, 0);
     update('value,10);
     add();
     if (lookup('sum) % 3 == 1) {
       pushNewDeclaration('value);
       update('value, 1);
       add();
       print("inner value = ", lookup('value));
       print("sum = ", lookup('sum));
       popDeclaration('value)
     }
     print("outer value = ", lookup('value));
   }
}
```

Object-oriented programming has scope for each object, so we have a nice controlled alternative to dynamic scoping (objects give names to scopes).

Good Practice for Scoping

- Static scoping is almost universally accepted in modern programming language design
- It is the approach that is usually easier to reason about and easier to **compile**, since we do not have names at compile time and compile each code piece separately
- Still, various ad-hoc language designs emerge and become successful
  - LISP implementations took dynamic scoping since it was simpler to implement for higher-order functions
  - Javascript

JavaScript

```
var fs = [];
for(var i = 0; i < 5; i++) {
  var c = i;
  fs.push(function() {
    console.log(c);
  });
}
for(var j = 0; j < 5; j++) {
  fs[j]();
}
```

can you guess what it will output?

How the **symbol map** changes in case of **static** scoping

Outer declaration
int value is shadowed by
inner declaration **string value**

Map becomes bigger as
we enter more scopes,
later becomes smaller again
Imperatively: need to make
maps bigger, later smaller again.
Functionally: immutable maps,
keep old versions.

```
class World {
 int sum;  int value;
 // value → int, sum → int
 void add(int foo) {
     // foo → int, value → int, sum → int
     string z;
     // z → string, foo → int, value → int, sum → int
     sum = sum + value; value = 0;
 }
 // value → int, sum → int
 void main(string bar) {
     // bar → string, value → int, sum → int
     int y;
     // y → int, bar → string, value → int, sum → int
     sum = 0;
     value = 10;
     add();
     // y → int, bar → string, value → int, sum → int
     if (sum % 3 == 1) {
       string value;
       // value → string, y → int, bar → string, sum → int
       value = 1;
       add();
       print("inner value = ", value);
       print("sum = ", sum); }
     // y → int, bar → string, value → int, sum → int
     print("outer value = ", value);
} }
```