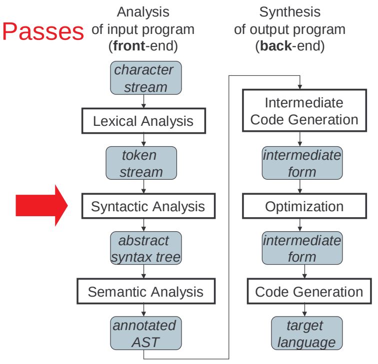
Syntactic Analysis

Syntactic analysis, or parsing, is the second phase of compilation: The token file is converted to an abstract syntax tree.

Compiler Passes



Analysis
of input program
(**front**-end)

Synthesis
of output program
(**back**-end)

character stream

Lexical Analysis

token stream

Syntactic Analysis

abstract syntax tree

Semantic Analysis

annotated AST

Intermediate Code Generation

intermediate form

Optimization

intermediate form

Code Generation

target language

Syntactic Analysis / Parsing

- Goal: Convert token stream to **abstract syntax tree**
- Abstract syntax tree (AST):
  - Captures the structural features of the program
  - Primary data structure for remainder of analysis
- Three Part Plan
  - Study how context-free grammars specify syntax
  - Study algorithms for parsing / building ASTs
  - Study the miniJava Implementation

Context-free Grammars

- Compromise between
  - REs, which can't nest or specify recursive structure
  - General grammars, too powerful, undecidable
- Context-free grammars are a sweet spot
  - Powerful enough to describe nesting, recursion
  - Easy to parse; but also allow restrictions for speed
- Not perfect
  - Cannot capture semantics, as in, "variable must be declared," requiring later semantic pass
  - Can be ambiguous
- EBNF, Extended Backus Naur Form, is popular notation

CFG Terminology

- **Terminals** -- alphabet of language defined by CFG
- **Nonterminals** -- symbols defined in terms of terminals and nonterminals
- **Productions** -- rules for how a nonterminal (lhs) is defined in terms of a (possibly empty) sequence of terminals and nonterminals
  - Recursion is allowed!
- Multiple productions allowed for a nonterminal, **alternatives**
- Start symbol -- root of the defining language

```
Program ::= Stmt
Stmt ::= if ( Expr ) then Stmt else Stmt
Stmt ::= while ( Expr ) do Stmt
```

EBNF Syntax of initial MiniJava

```
Program      ::= MainClassDecl { ClassDecl }
MainClassDecl ::= class ID {
                 public static void main
                 ( String [ ] ID ) { { Stmt } }
ClassDecl     ::= class ID [ extends ID ] {
                 { ClassVarDecl } { MethodDecl } }
ClassVarDecl  ::= Type ID ;
MethodDecl    ::= public Type ID
                 ( [ Formal { , Formal } ] )
                 { { Stmt } return Expr ; }
Formal        ::= Type ID
Type          ::= int |boolean | ID
```

Initial miniJava [continued]

```
Stmt ::= Type ID ;
       | { {Stmt} }
       | if ( Expr ) Stmt else Stmt
       | while ( Expr ) Stmt
       | System.out.println ( Expr ) ;
       | ID = Expr ;
Expr ::= Expr Op Expr
       | ! Expr
       | Expr . ID( [ Expr { , Expr } ] )
       | ID | this
       | Integer | true | false
       | ( Expr )
Op   ::= + | - | * | /
       | < | <= | >= | > | == | != | &&
```

RE Specification of initial MiniJava Lex

```
Program ::= (Token | Whitespace)*
Token ::= ID | Integer | ReservedWord | Operator |
           Delimiter
ID ::= Letter (Letter | Digit)*
Letter ::= a | ... | z | A | ... | Z
Digit ::= 0 | ... | 9
Integer ::= Digit⁺
ReservedWord::= class | public | static | extends |
        void | int | boolean | if | else |
       while|return|true|false| this | new | String
       | main | System.out.println
Operator ::= + | - | * | / | < | <= | >= | > | == |
       != | && | !
Delimiter ::= ; | . | , | = | ( | ) | { | } | [ | ]
```

Derivations and Parse Trees

Derivation: a sequence of expansion steps, beginning with a start symbol and leading to a sequence of terminals

Parsing: inverse of derivation

- Given a sequence of terminals (a\k\a tokens) want to recover the nonterminals representing structure

Can represent derivation as a **parse tree**, that is, the **concrete** syntax tree

Example Grammar

```
E  ::= E op E | - E | ( E ) | id
op ::= + | - | * | /
```

a * (b + - c)

Ambiguity

- Some grammars are **ambiguous**
  - Multiple distinct parse trees for the same terminal string
- Structure of the parse tree captures much of the meaning of the program
  - ambiguity implies multiple possible meanings for the same program

Famous Ambiguity: “Dangling Else”

```
Stmt ::= ... |
       if ( Expr ) Stmt |
       if ( Expr ) Stmt else Stmt
```

if (e_1) if (e_2) s_1 else s_2 : if (e_1) if (e_2) s_1 else s_2

Resolving Ambiguity

- Option 1: add a meta-rule
  - For example "`else` associates with closest previous `if`"
    - works, keeps original grammar intact
    - ad hoc and informal

Resolving Ambiguity [continued]

Option 2: rewrite the grammar to resolve ambiguity explicitly

```
Stmt          ::= MatchedStmt | UnmatchedStmt
MatchedStmt   ::= ... |
          if ( Expr ) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if ( Expr ) Stmt |
          if ( Expr ) MatchedStmt else UnmatchedStmt
```

- formal, no additional rules beyond syntax
- sometimes obscures original grammar

Resolving Ambiguity Example

```
Stmt          ::= MatchedStmt | UnmatchedStmt
MatchedStmt   ::= ... |
       if ( Expr ) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if ( Expr ) Stmt |
       if ( Expr ) MatchedStmt else UnmatchedStmt
```

`if (`e_1`)`   `if (`e_2`)` s_1 `else` s_2

Resolving Ambiguity [continued]

Option 3: redesign the language to remove the ambiguity

```
Stmt ::= ... |
          if Expr then Stmt end |
          if Expr then Stmt else Stmt end
```

- formal, clear, elegant
- allows sequence of `Stmts` in **then** and **else** branches, no { , } needed
- extra **end** required for every **if**

Another Famous Example

```
E  ::= E Op E | - E | ( E ) | id
Op ::= + | - | * | /
```

```
a   +   b   *   c   :   a   +   b   *   c
```

Resolving Ambiguity (Option 1)

Add some meta-rules, e.g. precedence and associativity rules

Example:

```
E ::= E Op E | - E | E ++
     | ( E ) | id
Op::=  + | - | * | / | %
     | ** | == | < | &&
     | ||
```

Operator	Preced	Assoc
Postfix ++	Highest	Left
Prefix -		Right
** (Exp)		Right
*, /, %		Left
+, -		Left
==, <		None
&&		Left
||	Lowest	Left

Removing Ambiguity (Option 2)

Option2: Modify the grammar to explicitly resolve the ambiguity

Strategy:

- create a nonterminal for each precedence level
- expr is lowest precedence nonterminal,
  each nonterminal can be rewritten with higher precedence operator, highest precedence operator includes atomic exprs
- at each precedence level, use:
  - left recursion for left-associative operators
  - right recursion for right-associative operators
  - no recursion for non-associative operators

Redone Example

```
E  ::= E0
E0 ::= E0 || E1 | E1                 left associative
E1 ::= E1 && E2 | E2                 left associative
E2 ::= E3 (== | <) E3 | E3           non associative
E3 ::= E3 (+ | -) E4 | E4            left associative
E4 ::= E4 (* | / | %) E5 | E5        left associative
E5 ::= E6 ** E5 | E6                 right associative
E6 ::= - E6 | E7                     right associative
E7 ::= E7 ++ | E8                    left associative
E8 ::= id | ( E )
```

Designing A Grammar

Concerns:

- Accuracy
- Unambiguity
- Formality
- Readability, Clarity
- Ability to be parsed by a particular algorithm:
  - Top down parser ==> LL(k) Grammar
  - Bottom up Parser ==> LR(k) Grammar
- Ability to be implemented using particular approach
  - By hand
  - By automatic tools

Parsing Algorithms

Given a grammar, want to parse the input programs

- Check legality
- Produce AST representing the structure
- Be efficient

- Kinds of parsing algorithms
  - Top down
  - Bottom up

Top Down Parsing

Build parse tree from the top (start symbol) down to leaves (terminals)

Basic issue:

- when "expanding" a nonterminal with some r.h.s., how to pick which r.h.s.?

E.g.

```
Stmts  ::= Call | Assign | If | While
Call   ::= Id ( Expr {,Expr} )
Assign ::= Id = Expr ;
If     ::= if Test then Stmts end
         | if Test then Stmts else Stmts end
While  ::= while Test do Stmts end
```

Solution: look at input tokens to help decide

Predictive Parser

Predictive parser: top-down parser that can select rhs by looking at most k input tokens (the **lookahead**)

Efficient:

- no backtracking needed
- linear time to parse

Implementation of predictive parsers:

- recursive-descent parser
  - each nonterminal parsed by a procedure
  - call other procedures to parse sub-nonterminals, recursively
  - typically written by hand
- table-driven parser
  - PDA:like table-driven FSA, plus stack to do recursive FSA calls
  - typically generated by a tool from a grammar specification

LL(k) Grammars

Can construct predictive parser automatically / easily if grammar is LL(k)

- Left-to-right scan of input, Leftmost derivation
- **k** tokens of look ahead needed, ≥ 1

Some restrictions:

- no ambiguity (true for any parsing algorithm)
- no **common prefixes** of length ≥ k:

```
If ::= if Test then Stmts end |
       if Test then Stmts else Stmts end
```

- no **left recursion**:

```
E  ::= E Op E | ...
```

- a few others

Restrictions guarantee that, given k input tokens, can always select correct rhs to expand nonterminal. Easy to do by hand in recursive-descent parser

Eliminating common prefixes

Can **left factor** common prefixes to eliminate them

- create new nonterminal for different suffixes
- delay choice till after common prefix

- Before:

```
If ::= if Test then Stmts end |
       if Test then Stmts else Stmts end
```

- After:

```
If     ::= if Test then Stmts IfCont
IfCont ::= end | else Stmts end
```

Eliminating Left Recursion

- Can Rewrite the grammar to eliminate left recursion
- Before

```
E ::= E + T | T
T ::= T * F | F
F ::= id | ...
```

- After

```
E    ::= T ECon
ECon ::= + T ECon | ε
T    ::= F TCon
TCon ::= * F TCon | ε
F    ::= id | ...
```

Bottom Up Parsing

Construct parse tree for input from leaves up

- reducing a string of tokens to single start symbol (inverse of deriving a string of tokens from start symbol)

“Shift-reduce” strategy:

- read (“shift”) tokens until seen r.h.s. of “correct” production `xyzabcdef` A ::= **bc**.D

```
xyzabcdef
     ^
```

- reduce handle to l.h.s. nonterminal, then continue
- done when all input read and reduced to start nonterminal

LR(k)

- LR(k) parsing
  - **L**eft-to-right scan of input, **R**ightmost derivation
  - ***k*** tokens of look ahead
- Strictly more general than LL(*k*)
  - Gets to look at whole rhs of production before deciding what to do, not just first k tokens of rhs
  - can handle left recursion and common prefixes fine
  - Still as efficient as any top-down or bottom-up parsing method
- Complex to implement
  - need automatic tools to construct parser from grammar

LR Parsing Tables

Construct parsing tables implementing a FSA with a stack

- rows: states of parser
- columns: token(s) of lookahead
- entries: action of parser
  - shift, goto state X
  - reduce production “`X ::= RHS`”
  - accept
  - error

Algorithm to construct FSA similar to algorithm to build DFA from NFA

- each state represents set of possible places in parsing

LR(k) algorithm builds huge tables

LALR-Look Ahead LR

LALR(*k*) algorithm has fewer states ==> smaller tables

- less general than LR(*k*), but still good in practice
- size of tables acceptable in practice

- k == 1 in practice
  - most parser generators, including `yacc` and `jflex`, are LALR(1)

Global Plan for LR(0) Parsing

- Goal: Set up the tables for parsing an LR(0) grammar
  - Add S' --> S$ to the grammar, i.e. solve the problem for a new grammar with terminator
  - Compute parser states by starting with state 1 containing added production, S' --> .S$
  - Form closures of states and shifting to complete diagram
  - Convert diagram to transition table for PDA
  - Step through parse using table and stack

LR(0) Parser Generation

Example grammar:

```
S′ ::= S $       // always add this production
S ::= beep | { L }
L ::= S | L ; S
```

- Key idea: simulate where input might be in grammar as it reads tokens
- "Where input might be in grammar" captured by set of items, which forms a state in the parser's FSA
  - LR(0) item: `lhs ::= rhs` production, with dot in rhs somewhere marking what's been read (shifted) so far
    - LR(k) item: also add *k* tokens of lookahead to each item
  - Initial item: `S′ ::= . S $`

Closure

Initial state is **closure** of initial item

- closure: if dot before non-terminal, add all productions for that non-terminal with dot at the start
  - "epsilon transitions"

Initial state (1):

```
S′::= . S $
S ::= . beep
S ::= . { L }
```

State Transitions

Given set of items, compute new state(s) for each symbol (terminal and non-terminal) after dot

- state transitions correspond to shift actions

New item derived from old item by shifting dot over symbol

- do closure to compute new state Initial state (1):
  `S′ ::= . S $ S ::= . beep S ::= .{ L }`
- State (2) reached on transition that shifts `S`:
  `S′ ::= S . $`
- State (3) reached on transition that shifts **beep**:
  `S ::= beep .`
- State (4) reached on transition that shifts **{**:

```
S ::= { . L }
L ::= . S
L ::= . L ; S
S ::= . beep
S ::= . { L }
```

Accepting Transitions

If state has `S′ ::= ... . $` item,
then add transition labeled$ to the accept action

Example:

`S′ ::= S . $`

has transition labeled $ to `accept` action

Reducing States

If state has *`lhs ::= rhs .`* item, then it has a `reduce` *`lhs ::= rhs`* action

Example:

`S ::= `**`beep`**` .`

has `reduce S ::= `**`beep`** action

No label; this state always reduces this production

- what if other items in this state shift, or accept?
- what if other items in this state reduce differently?

Rest of the States, Part 1

State (4): if shift **beep**, goto State (3)
State (4): if shift **{**, goto State (4)
State (4): if shift S, goto State (5)
State (4): if shift L, goto State (6)

State (5):

`L ::= S .`

State (6):

`S ::= `**`{`**` L . `**`}`**
`L ::= L . ; S`

State (6): if shift **}**, goto State (7)
State (6): if shift **;**, goto State (8)

Rest of the States (Part 2)

State (7):

`S ::= `**`{`**` L `**`}`**` .`

State (8):

`L ::= L ; . S`
`S ::= . `**`beep`**
`S ::= . `**`{`**` L `**`}`**

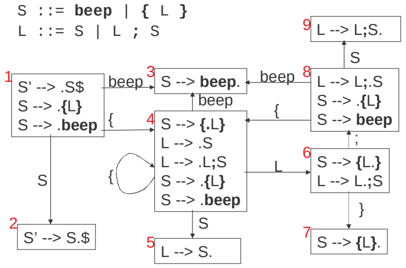
State (8): if shift **beep**, goto State (3)
State (8): if shift **{**, goto State (4)
State (8): if shift S, goto State (9)

State (9):

`L ::= L ; S .` (whew)

LR(0) State Diagram

`S′::= S $`
`S ::= `**`beep`**` | `**`{`**` L `**`}`**
`L ::= S | L ; S`



Building Table of States & Transitions

Create a row for each state
Create a column for each terminal, non-terminal, and $
For every "state (*i*): if shift *X* goto state (*j*)" transition:

- if *X* is a terminal, put "shift, goto *j*" action in row *i*, column *X*
- if *X* is a non-terminal, put "goto *j*" action in row *i*, column *X*

For every "state (*i*): if $ accept" transition:

- put "accept" action in row *i*, column $

For every "state (*i*): *`lhs ::= rhs.`*" action:

- put "`reduce` *`lhs ::= rhs`*" action in all columns of row *i*

Table of This Grammar

State	{	}	beep	;	S	L	$
1	s,g4		s,g3		g2		
2							a!
3	`reduce S ::= beep`						
4	s,g4		s,g3		g5	g6	
5	`reduce L ::= S`						
6		s,g7		s,g8			
7	`reduce S ::= { L }`						
8	s,g4		s,g3		g9		
9	`reduce L ::= L ; S`						

Example

```
S'::= S $
S ::= beep | { L }
L ::= S | L ; S
```

St	{	}	beep	;	S	L	$
1	s,g4		s,g3		g2		
2							a!
3	`reduce S ::= beep`						
4	s,g4		s,g3		g5	g6	
5	`reduce L ::= S`						
6		s,g7		s,g8			
7	`reduce S ::= { L }`						
8	s,g4		s,g3		g9		
9	`reduce L ::= L ; S`						

1	{ beep ; { beep } } $
1 { 4	beep ; { beep } } $
1 { 4 beep 3	; { beep } } $
1 { 4 S 5	; { beep } } $
1 { 4 L 6	; { beep } } $
1 { 4 L 6 ; 8	{ beep } } $
1 { 4 L 6 ; 8 { 4	beep } } $
1 { 4 L 6 ; 8 { 4 beep 3	} } $
1 { 4 L 6 ; 8 { 4 S 5	} } $
1 { 4 L 6 ; 8 { 4 L 6	} } $
1 { 4 L 6 ; 8 { 4 L 6 } 7	} $
1 { 4 L 6 ; 8 S 9	} $
1 { 4 L 6	} $
1 { 4 L 6 } 7	$
1 S 2	$
accept	

Problems In Shift-Reduce Parsing

Can write grammars that cannot be handled with shift-reduce parsing

Shift/reduce conflict:

- state has both shift action(s) and reduce actions

Reduce/reduce conflict:

- state has more than one reduce action

Shift/Reduce Conflicts

LR(0) example:

`E ::= E + T | T`

State: `E ::= E . + T`
`E ::= T .`

- Can shift `+`
- Can reduce `E ::= T`

LR(k) example:

```
S ::= if E then S |
      if E then S else S | ...
```

State: `S ::= if E then S .`
`S ::= if E then S . else S`

- Can shift `else`
- Can reduce `S ::= if E then S`

Avoiding Shift-Reduce Conflicts

Can rewrite grammar to remove conflict

- E.g. `Matched Stmt vs. Unmatched Stmt`

Can resolve in favor of shift action

- try to find longest r.h.s. before reducing
  - works well in practice
  - `yacc`, `jflex`, et al. do this

Reduce/Reduce Conflicts

Example:

```
Stmt ::= Type id ; | LHS = Expr ; | ...
...
LHS ::= id | LHS [ Expr ] | ...
...
Type ::= id | Type [] | ...
```

State: `Type ::= id .`
`LHS ::= id .`

Can reduce `Type ::= id`

Can reduce `LHS ::= id`

Avoid Reduce/Reduce Conflicts

Can rewrite grammar to remove conflict

- can be hard
  - e.g. C/C++ declaration vs. expression problem
  - e.g. MiniJava array declaration vs. array store problem

Can resolve in favor of one of the reduce actions

- but which?
- `yacc`, `jflex`, et al. Pick reduce action for production listed textually first in specification

Abstract Syntax Trees

The parser’s output is an abstract syntax tree (AST) representing the grammatical structure of the parsed input

- ASTs represent only semantically meaningful aspects of input program, unlike concrete syntax trees which record the complete textual form of the input
  - There’s no need to record keywords or punctuation like `()`, `;`, `else`
  - The rest of compiler only cares about the abstract structure

AST Node Classes

Each node in an AST is an instance of an AST class

- `IfStmt`, `AssignStmt`, `AddExpr`, `VarDecl`, etc.

Each AST class declares its own instance variables holding its AST subtrees

- `IfStmt` has `testExpr`, `thenStmt`, and `elseStmt`
- `AssignStmt` has `lhsVar` and `rhsExpr`
- `AddExpr` has `arg1Expr` and `arg2Expr`
- `VarDecl` has `typeExpr` and `varName`

AST Class Hierarchy

AST classes are organized into an inheritance hierarchy based on commonalities of meaning and structure

- Each "abstract non-terminal" that has multiple alternative concrete forms will have an abstract class that’s the superclass of the various alternative forms
  - `Stmt` is abstract superclass of `IfStmt`, `AssignStmt`, etc.
  - `Expr` is abstract superclass of `AddExpr`, `VarExpr`, etc.
  - `Type` is abstract superclass of `IntType`, `ClassType`, etc.

AST Extensions For Project

New variable declarations:

- `StaticVarDecl`

New types:

- `DoubleType`
- `ArrayType`

New/changed statements:

- `IfStmt` can omit else branch
- `ForStmt`
- `BreakStmt`
- `ArrayAssignStmt`

New expressions:

- `DoubleLiteralExpr`
- `OrExpr`
- `ArrayLookupExpr`
- `ArrayLengthExpr`
- `ArrayNewExpr`

Automatic Parser Generation in MiniJava

We use the CUP tool to automatically create a parser from a specification file, `Parser/minijava.cup`

The MiniJava Makefile automatically rebuilds the parser whenever its specification file changes

A CUP file has several sections:

- introductory declarations included with the generated parser
- declarations of the terminals and nonterminals with their types
- The AST node or other value returned when finished parsing that nonterminal or terminal
- precedence declarations
- productions + actions

Terminal and Nonterminal Declarations

Terminal declarations we saw before:

```
/* reserved words: */
terminal CLASS, PUBLIC, STATIC, EXTENDS;
...
/* tokens with values: */
terminal String IDENTIFIER;
terminal Integer INT_LITERAL;
```

Nonterminals are similar:

```
nonterminal Program Program;
nonterminal MainClassDecl MainClassDecl;
nonterminal List/*<...>*/ ClassDecls;
nonterminal RegularClassDecl ClassDecl;
...
nonterminal List/*<Stmt>*/ Stmts;
nonterminal Stmt Stmt;
nonterminal List/*<Expr>*/ Exprs;
nonterminal List/*<Expr>*/ MoreExprs;
nonterminal Expr Expr;
nonterminal String Identifier;
```

Precedence Declarations

Can specify precedence and associativity of operators

- equal precedence in a single declaration
- lowest precedence textually first
- specify left, right, or nonassoc with each declaration

Examples:

```
precedence left AND_AND;
precedence nonassoc EQUALS_EQUALS,
                     EXCLAIM_EQUALS;
precedence left LESSTHAN, LESSEQUAL,
               GREATEREQUAL, GREATERTHAN;
precedence left PLUS, MINUS;
precedence left STAR, SLASH;
precedence left EXCLAIM;
precedence left PERIOD;
```

Productions

All of the form:

```
LHS ::=  RHS1 {: Java code 1 :}
       | RHS2 {: Java code 2 :}
       | ...
       | RHSn {: Java code n :};
```

Can label symbols in RHS with `:var` suffix to refer to its result value in Java code

- `varleft` is set to line in input where var symbol was

E.g.: `Expr ::= Expr:arg1 PLUS Expr:arg2`

```
{: RESULT = new AddExpr( arg1,arg2,arg1left);:}
| INT_LITERAL:value{: RESULT = new IntLiteralExpr(
      value.intValue(),valueleft);:}
| Expr:rcvr PERIOD Identifier:message OPEN_PAREN
      Exprs:args CLOSE_PAREN
{: RESULT = new MethodCallExpr(
      rcvr,message,args,rcvrleft);:}
| ... ;
```

Error Handling

How to handle syntax error?

Option 1: quit compilation

+ easy

- inconvenient for programmer

Option 2: error recovery

+ try to catch as many errors as possible on one compile

- difficult to avoid streams of spurious errors

Option 3: error correction

+ fix syntax errors as part of compilation

- hard!!

Panic Mode Error Recovery

When finding a syntax error, skip tokens until reaching a "landmark"

- landmarks in MiniJava: **;**, **)**, **}**
- once a landmark is found, hope to have gotten back on track

In top-down parser, maintain set of landmark tokens as recursive descent proceeds

- landmarks selected from terminals later in production
- as parsing proceeds, set of landmarks will change, depending on the parsing context

In bottom-up parser, can add special error nonterminals, followed by landmarks

- if syntax error, then will skip tokens till seeing landmark, then reduce and continue normally

- E.g.

```
Stmt ::= ... | error ; | { error }
Expr ::= ... | ( error )
```