Contents

- 2.1 Grammars
  - 2.1.1 Backus-Naur Form
  - 2.1.2 Derivations
  - 2.1.3 Parse Trees
  - 2.1.4 Associativity and Precedence
  - 2.1.5 Ambiguous Grammars
- 2.2 Extended BNF
- 2.3 Syntax of a Small Language: *Clite*
  - 2.3.1 Lexical Syntax
  - 2.3.2 Concrete Syntax
- 2.4 Compilers and Interpreters
- 2.5 Linking Syntax and Semantics
  - 2.5.1 Abstract Syntax
  - 2.5.2 Abstract Syntax Trees
  - 2.5.3 Abstract Syntax of *Clite*

2.3 Syntax of a Small Language: *Clite*

- Motivation for using a subset of C:

Language	Grammar (pages)	Reference
Pascal	5	Jensen & Wirth
C	6	Kernighan & Richie
C++	22	Stroustrup
Java	14	Gosling, et. al.

- The *Clite* grammar fits on one page (or 3 slides)

Fig. 2.7 *Clite* Grammar: Statements

Program → `int main ( )` { *Declarations Statements* }
Declarations → { *Declaration* }
Declaration → *Type Identifier* [[*Integer*]] { , *Identifier* [[*Integer*]] }
Type → `int` | `bool` | `float` | `char`
Statements → { *Statement* }
Statement → `;` | *Block* | *Assignment* | *IfStatement* | *WhileStatement*
Block → { *Statements* }
Assignment → *Identifier* [[*Expression*]] = *Expression* `;`
IfStatement → `if ( ` *Expression* `)` *Statement* [`else` *Statement*]
WhileStatement → `while ( ` *Expression* `)` *Statement*

Fig. 2.7 *Clite* Grammar: Expressions

Expression → *Conjunction* { `||` *Conjunction* }
Conjunction → *Equality* { `&&` *Equality* }
Equality → *Relation* [*EquOp Relation*]
EquOp → `==` | `!=`
Relation → *Addition* [*RelOp Addition*]
RelOp → `<` | `<=` | `>` | `>=`
Addition → *Term* { *AddOp Term* }
AddOp → `+` | `-`
Term → *Factor* { *MulOp Factor* }
MulOp → `*` | `/` | `%`
Factor → [*UnaryOp*] *Primary*
UnaryOp → `-` | `!`
Primary → *Identifier* [`[` *Expression* `]`] | *Literal* | `(` *Expression* `)` |
Type `(` *Expression* `)`

Fig. 2.7 *Clite* grammar: lexical level

Identifier → *Letter* { *Letter* | *Digit* }
Letter → `a` | `b` | ... | `z` | `A` | `B` | ... | `Z`
Digit → `0` | `1` | ... | `9`
Literal → *Integer* | *Boolean* | *Float* | *Char*
Integer → *Digit* { *Digit* }
Boolean → `true` | `false`
Float → *Integer* `.` *Integer*
Char → `'` *ASCII Char* `'`

Issues Not Addressed by this Grammar

- Comments
- Whitespace
- Distinguishing one token <= from two tokens < =
- Distinguishing identifiers from keywords like if
- These issues are addressed by identifying two levels:
  - *lexical level*
  - *syntactic level*

2.3.1 Lexical Syntax

- *Input*: a stream of characters from the ASCII set, keyed by a programmer.
- *Output*: a stream of *tokens* or basic symbols, classified as follows:
  - *Identifiers* e.g., Stack, x, i, push
  - *Literals* e.g., 123, 'x', 3.25, true
  - *Keywords* bool char else false float if int main true while
  - *Operators* = || && == != < <= > >= + - * / !
  - *Punctuation* ; , { } ( )

Whitespace

- Whitespace is any space, tab, end-of-line character (or characters), or character sequence inside a comment
- No token may contain embedded whitespace (unless it is a character or string literal)
- Example:

  \>= *one token*

  \> = *two tokens*

Whitespace Examples in Pascal

- while a < b do *legal* - spacing between tokens
- while a<b do spacing not needed for <
- whilea<bdo *illegal* - can't tell boundaries
- whilea < bdo between tokens

Comments

- Not defined in grammar
- *Clite* uses // comment style of C++

Identifier

- Sequence of letters and digits, starting with a letter
- *if is both an identifier and a keyword*
- *Most languages require identifiers to be distinct from keywords*
- In some languages, identifiers are merely predefined (and thus can be redefined by the programmer)

Redefining Identifiers

```
program confusing;
const true = false;
begin
    if  (a<b) = true then
    f(a)
    else …
```

Should Identifiers be case-sensitive?

- Older languages: no. Why?
- Pascal, Ada, most BASICs: no.
- Modula, C, C++, Java: yes
- PHP: partly yes (variables), partly no (keywords, function names).
- Smalltalk, Haskell, Prolog: capitalization encodes semantic information

2.3.2 Concrete Syntax

- Based on a parse of its *Tokens*
- *; is a statement terminator*
- *(Algol-60, Pascal use ; as a separator)*
- Rule for *IfStatement* is ambiguous:

  "The else ambiguity is resolved by connecting an **else** with the last encountered else-less if."

  [Stroustrup, 1991]

Expressions in *Clite*

- 13 grammar rules
- Use of meta braces – operators are left associative
- Note use of meta-brackets for *Relation* and *Equality*
  - These are non-associative
- C++ expressions require 4 pages of grammar rules [Stroustrup]
- C uses an ambiguous expression grammar [Kernighan and Ritchie]
  - *What does* `x * y ;` *mean?*

Associativity and Precedence

Clite Operator	Associativity
Unary - !	none
* /	left
+ -	left
< <= > >=	none
== !=	none
&&	left
||	left

Clite Equality, Relational Operators

- … are non-associative.
- (an idea borrowed from Ada)
- Why is this important?

  In C++, the expression:

  if (a < x < b)

  is *not* equivalent to

  if (a < x && x < b)

  But it is error-free!

  So, what does it mean?

Precedence of Operators in C++

Precedence	Operator	Description	Example	Associativity
1	::	Scoping operator	Class::age = 2;	none
2	()	Grouping operator	(a + b) / 4;	left to right
	[]	Array access	array[4] = 2;	
	->	Member access from a pointer	ptr->age = 34;	
	.	Member access from an object	obj.age = 34;	
	++	Post-increment	for(i = 0; i < 10; i++) ...	
	--	Post-decrement	for(i = 10; i > 0; i--) ...	
3	!	Logical negation	if(!done) ...	right to left
	~	Bitwise complement	flags = ~flags;	
	++	Pre-increment	for(i = 0; i < 10; ++i) ...	
	--	Pre-decrement	for(i = 10; i > 0; --i) ...	
	-	Unary minus	int i = -1;	
	+	Unary plus	int i = +1;	
	*	Dereference	data = *ptr;	
	&	Address of	address = &obj;	
	(type)	Cast to a given type	int i = (int) floatNum;	
	sizeof	Return size in bytes	int size = sizeof(floatNum);	
4	->*	Member pointer selector	ptr->*var = 24;	left to right
	.*	Member object selector	obj.*var = 24;	
5	*	Multiplication	int i = 2 * 4;	left to right
	/	Division	float f = 10 / 3;	
	%	Modulus	int rem = 4 % 3;	

Precedence of Operators in C++ (2)

6	+ -	Addition Subtraction	int i = 2 + 3; int i = 5 - 1;	left to right
7	<< >>	Bitwise shift left Bitwise shift right	int flags = 33 << 1; int flags = 33 >> 1;	left to right

8	<
	 <=
	 >
	 >=
		Comparison less-than
		 Comparison less-than-or-equal-to
		 Comparison greater-than
		 Comparison geater-than-or-equal-to
			if(i < 42) ...
			 if(i <= 42) ...
			 if(i > 42) ...
			 if(i >= 42) ...
				left to right

9	== !=	Comparison equal-to Comparison not-equal-to	if(i == 42) ... if(i != 42) ...	left to right
10	&	Bitwise AND	flags = flags & 42;	left to right
11	^	Bitwise exclusive OR	flags = flags ^ 42;	left to right
12	|	Bitwise inclusive (normal) OR	flags = flags | 42;	left to right

Precedence of Operators in C++ (3)

13	&&	Logical AND	if(conditionA && conditionB) ...	left to right
14	||	Logical OR	if(conditionA || conditionB) ...	left to right
15	? :	Ternary conditional (if-then-else)	int i = (a > b) ? a : b;	right to left

16	=
	 +=
	 -=
	 *=
	 /=
	 %=
	 &=
	 ^=
	 |=
	 <<=
	 >>=
		Assignment operator
		 Increment and assign
		 Decrement and assign
		 Multiply and assign
		 Divide and assign
		 Modulo and assign
		 Bitwise AND and assign
		 Bitwise exclusive OR and assign
		 Bitwise inclusive (normal) OR and assign
		 Bitwise shift left and assign
		 Bitwise shift right and assign
			int a = b;
			 a += 3;
			 b -= 4;
			 a *= 5;
			 a /= 2;
			 a %= 3;
			 flags &= new_flags;
			 flags ^= new_flags;
			 flags |= new_flags;
			 flags <<= 2;
			 flags >>= 2;
				right to left

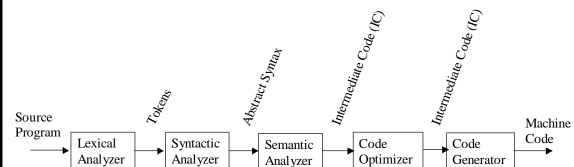
17	,	Sequential evaluation operator	for(i = 0, j = 0; i < 10; i++, j++) ...	left to right

Order of evaluation / side-effects not specified

- What is the meaning of this expression?

```
int x = 1;
x = x / ++x;
```

2.4 Compilers and Interpreters



Lexical analyzer (Lexer)

- Input: characters
- Output: tokens
- Separate from because:
  - *Speed: 75% of time for non-optimizing compilers*
  - *Simpler design (DFSA)*
  - *Historical: handled machine-specific character sets, OS-specific end-of-line conventions*
    - Mac prior to OS/X: ASCII 13
    - Unix: ASCII 10
    - Windows: 13 10

Syntactic Analyzer (Parser)

- Based on BNF/EBNF grammar
- Input: tokens
- Output: abstract syntax tree (parse tree)
- Abstract syntax: parse tree with punctuation, many nonterminals discarded

Semantic Analysis (Type Checker)

- Input: Abstract syntax tree
- Output: Intermediate Code (IC) tree
- Responsible for enforcing compile-time semantic rules:
  - *Check that all identifiers are declared*
  - *Perform type checking*
  - *Insert implied conversion operators (i.e., make them explicit)*
- IC tree might have type-specific operators, etc.

Code Optimization

- Improve intermediate code
  - *Both machine independent and machine dependent optimizations may be used*
- Evaluate constant expressions at compile-time
- Reorder code to improve cache performance
- Reorder code for superscalar architectures
- Eliminate common subexpressions
- Eliminate unnecessary code

Example CL Optimization switches

```
/O1 minimize space
/O2 maximize speed
/Oa assume no aliasing
/Ob<n> inline expansion (default n=0)
/Od disable optimizations (default)
/Og enable global optimization
/Oi enable intrinsic functions
/Op[-] improve floating-pt consistency
/Os favor code space
/Ot favor code speed
/Ow assume cross-function aliasing
/Ox maximum opts. (/Ogityb1 /Gs)
/Oy[-] enable frame pointer omission
```

Example gcc Optimization Switches

Table 5-1. *Optimizations Enabled with -O and -O1*

Optimization	Description
-fcprop-registers	Attempts to reduce the number of register copy operations performed.
-fdefer-pop	Accumulates function arguments on the stack.
-fdelayed-branch	Utilizes instruction slots available after delayed branch instructions.
-fguess-branch-probability	Uses a randomized predictor to guess branch possibilities.
-fif-conversion	Converts conditional jumps into nonbranching code.
-fif-conversion2	Performs if-conversion using conditional execution (on CPUs that support it).
-floop-optimize	Applies several loop-specific optimizations.
-fmerge-constants	Merges identical constants used in multiple modules.
-fomit-frame-pointer	Omits storing function frame pointers in a register. Only activated on systems where this does not interfere with debugging.
-ftree-ccp	Performs sparse conditional constant propagation (CCP) on SSA trees (GCC 4.x only).
-ftree-ch	Performs loop header copying on SSA trees, which eliminates a jump and provides opportunities for subsequent code motion optimization (GCC 4.x only).
-ftree-copyrename	Performs copy renaming on SSA trees, which attempts to rename internal compiler temporary names at copy location to names that more closely resemble the original variable names (GCC 4.x only).

Code Generation

- Construct target machine code
- Output: machine code
  - *Instruction selection*
  - *Register management*
  - *Peephole optimization:*

Constant Folding: Evaluate constant subexpressions in advance.
Strength Reduction: Replace slow operations with faster equivalents.
Null Sequences: Delete useless operations
Combine Operations: Replace several operations with one equivalent.
Algebraic Laws: Use algebraic laws to simplify or reorder instructions.
Special Case Instructions: Use instructions designed for special operand cases.
Address Mode Operations: Use address modes to simplify code.

Interpreter

- Replaces last 2 phases of a compiler
- Input:
  - *Mixed: intermediate code*
  - *Pure: stream of ASCII characters*
- Mixed interpreters
  - *Java, Perl, Python, Haskell, Scheme*
- Pure interpreters:
  - *most Basics, shell commands*
- Just-In-Time (JIT) interpreters
  - *Intermediate bytecode compiled to native machine code and cached during execution*

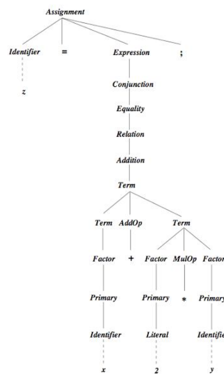
Just-In-Time (JIT) Compilers

- Source code is compiled to bytecode
- Intermediate bytecode is compiled to native machine code and cached during execution
- Widely used today:
  - *.NET*
  - *Java*
  - *Actionscript*
- Example: Java programs are compiled by *javac* to byte code. Interpreter is *java*. In recent versions compiles JIT code on most platforms.

2.5 Linking Syntax and Semantics

- Output: parse tree is inefficient
- Example: Fig. 2.9

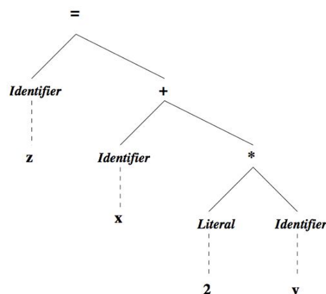
Parse Tree for
z = x + 2*y;
Fig. 2.9



Finding a More Efficient Tree

- The *shape* of the parse tree reveals the meaning of the program.
- So we want a tree that removes its inefficiency and keeps its shape.
  - *Remove separator/punctuation terminal symbols*
  - *Remove all trivial root (single child) nonterminals*
  - *Replace remaining nonterminals with terminals (typically operators) that are leaves of immediate subtrees*
- Example: Fig. 2.10

Abstract Syntax Tree for
z = x + 2*y;
Fig. 2.10



Abstract Syntax

Removes "syntactic sugar" and keeps essential elements of a language. E.g., consider the following two equivalent loops:

- Pascal

```
while i < n do begin
   i := i + 1;
end;
```

- C/C++

```
while (i < n) {
   i = i + 1;
}
```

Essential information
1) it is a *loop*
2) its terminating condition is i < n
3) its body increments the current value of i.

Abstract Syntax Rules

- Lhs = Rhs, where Lhs is the name of an abstract syntactic class and Rhs is one of:

1. A list of one or more alternatives

   example: expressions are identifiers, literals, expr with a unary op, or expr with a binary op

2. A list of essential components that define a member of that class (properties, fields, etc.)

   example: assignment has target and source

Abstract Syntax of *Clite* Assignments

•*Assignment = Variable* target*; Expression* source
•*Expression = VariableRef | Value | Binary | Unary*
•*VariableRef = Variable | ArrayRef*
•*Variable = String* id
•*ArrayRef = String* id*; Expression* index
•*Value = IntValue | BoolValue | FloatValue | CharValue*
•*Binary = Operator* op*; Expression* term1, term2
•*Unary = UnaryOp* op*; Expression* term
•*Operator = ArithmeticOp | RelationalOp | BooleanOp*
•*IntValue = Integer* intValue
•…

Abstract Syntax and OO Classes

- Abstract syntax can directly provide a class hierarchy in an object-oriented language
- Instance variable definitions are taken from the abstract syntax definition
- Some classes (e.g. Expression) are abstract classes because we can only have instances of a subtype
- Note that the actual programming language disappears when translated into abstract syntax
  - *Similar to object code at the machine level*
  - *Consider .NET langs: VB, C#, Jscript, etc.*

Abstract Syntax as Java Classes

```
abstract class Expression { }
abstract class VariableRef extends Expression { }
class Variable extends VariableRef { String id; }
class Value extends Expression { … }
class Binary extends Expression {
   Operator op;
   Expression term1, term2;
}
class Unary extends Expression {
   UnaryOp op;
   Expression term;
}
```

Abstract syntax trees

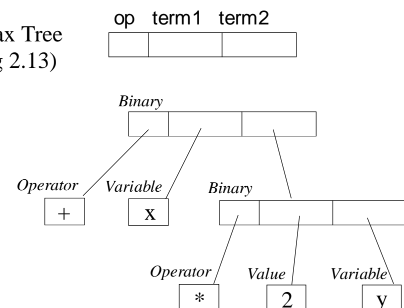
- Concrete syntax provides form; abstract syntax provides definition of essential elements
- An abstract syntax tree has nodes for each syntactic category
  - *Each node has fields for RHS elements*
  - *Some nodes such as Expression are a grouping mechanism only; no associated fields*

Example Abstract Syntax Tree

- *Binary* node

op term1 term2

Abstract Syntax Tree for x+2*y (Fig 2.13)



Basic Abstract Syntax of *Clite*
(*Declarations* and *Statements*)
Fig 2.14

Program = *Declarations* `decpart;` *Statements* `body;`
Declarations = *Declaration**
Declaration = *VariableDecl* | *ArrayDecl*
VariableDecl = *Variable* `v;` *Type* `t`
ArrayDecl = *Variable* `v;` *Type* `t;` *Integer* `size`
Type = `int` | `bool` | `float` | `char`
Statements = *Statement**
Statement = *Skip* | *Block* | *Assignment* | *Conditional* | *Loop*
Skip =
Block = *Statements*
Conditional = *Expression* `test;` *Statement* `thenbranch, elsebranch`
Loop = *Expression* `test;` *Statement* `body`