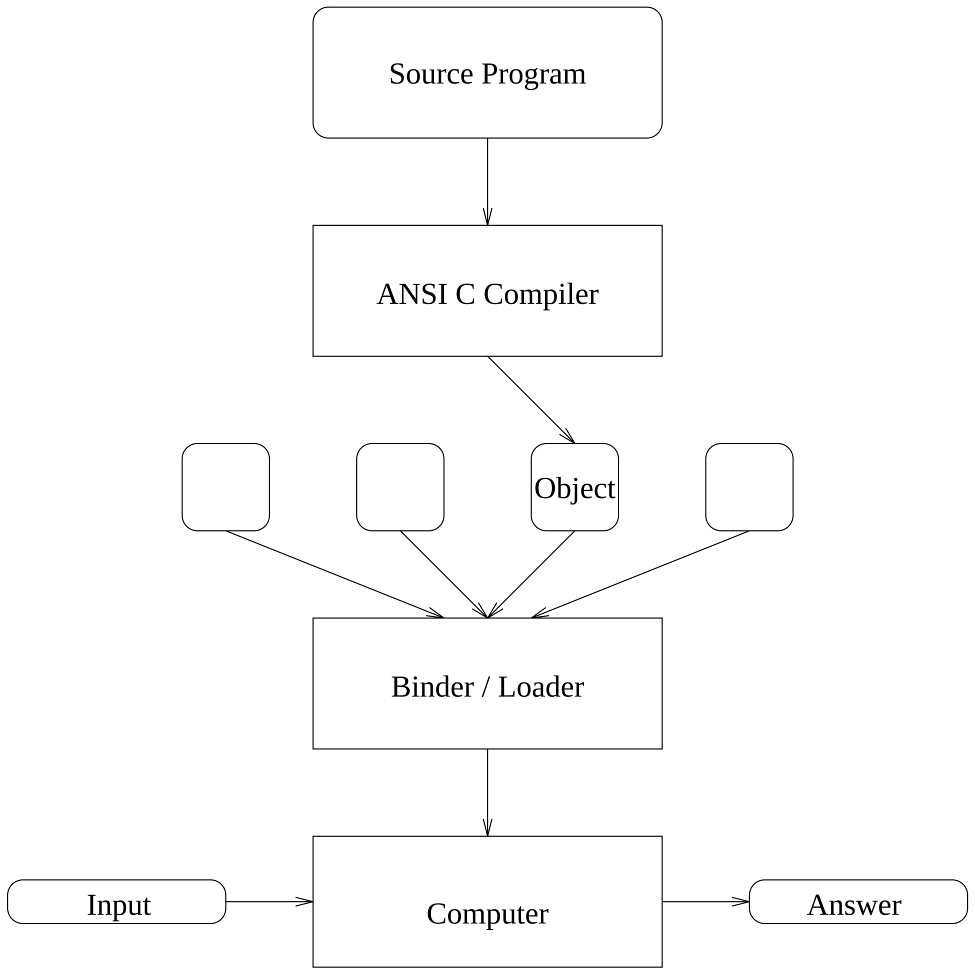
Tutorial outline

What is a language translator?

You type: cc foo.c. . . What happens?



Language: Vehicle (architecture) for transmitting information between components of a system. For our purposes, a language is a *formal interface*. The goal of every compiler is correct and efficient language translation.

The process of language translation

1. **A person has an idea of how to compute something:**

$$fact(n) = \begin{cases} 1 & \textbf{if } n \leq 0 \\ n \times fact(n-1) & \textbf{otherwise} \end{cases}$$

2. **An algorithm captures the essence of the computation:**

$$fact(n) = \textbf{if } n \leq 0 \textbf{ then } 1 \textbf{ else } n \times fact(n-1)$$

   Typically, a *pseudocode* **language is used, such as “pidgin ALGOL”.**

3. **The algorithm is expressed in some programming language:**

```
int fact(int n) {
   if (n <= 0) return(1);
   else return(n*fact(n-1));
}
```

We would be done if we had a computer that “understood” the language directly. So why don’t we build more C machines?

a) **How does the machine know it’s seen a C program and not a Shakespeare sonnet?**

b) **How does the machine know what is “meant” by the C program?**

c) **It’s hard to build such machines. What happens when language extensions are introduced (C++)?**

d) **RISC philosophy says simple machines are better.**

Finally...

A compiler translates programs written in a *source* language into a *target* language. For our purposes, the source language is typically a *programming language*—convenient for humans to use and understand—while the target language is typically the (relatively low-level) instruction set of a computer.

Source Program

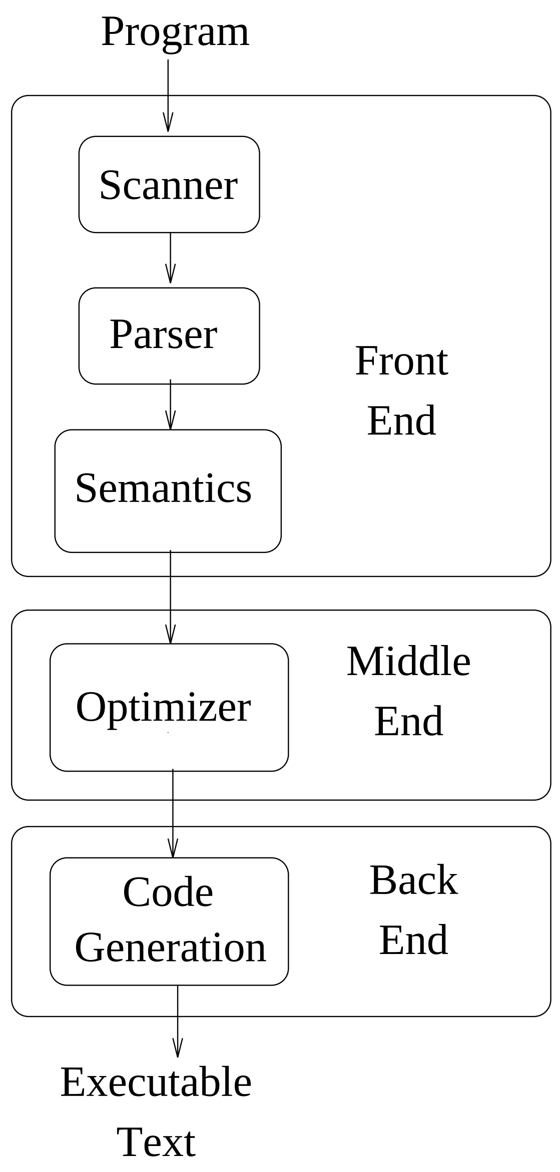
```
main() {
   int a;

   a += 5.0;
}
```

Target Program (Assembly)

```
_main:
    !#PROLOGUE# 0
    sethi   %hi(LF12),%g1
    add %g1,%lo(LF12),%g1
    save    %sp,%g1,%sp
    !#PROLOGUE# 1
    sethi   %hi(L2000000),%o0
    ldd [%o0+%lo(L2000000)],%f0
    ld  [%fp+-0x4],%f2
    fitod   %f2,%f4
    faddd   %f4,%f0,%f6
    fdtoi   %f6,%f7
    st  %f7,[%fp+-0x4]
```

Running the Sun `cc` compiler on the above source program of 32 characters produces the assembly program shown to the right. The bound binary executable occupied in excess of 24 thousand bytes.

Structure of a compiler

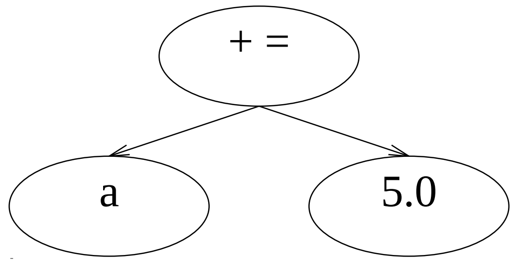


Front End

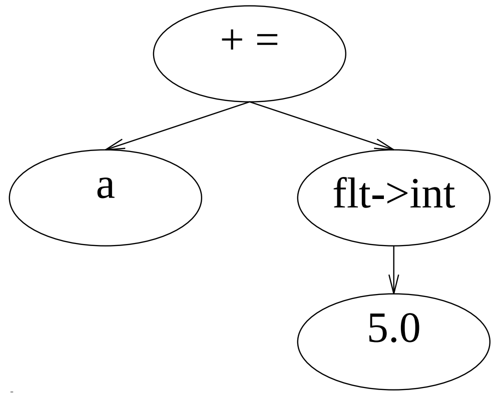
Scanner: decomposes the input stream into *tokens*. So the string "a += 5.0;" becomes

$\boxed{a}\ \boxed{+=}\ \boxed{5.0}\ \boxed{;}$

Parser: analyzes the tokens for correctness and structure:

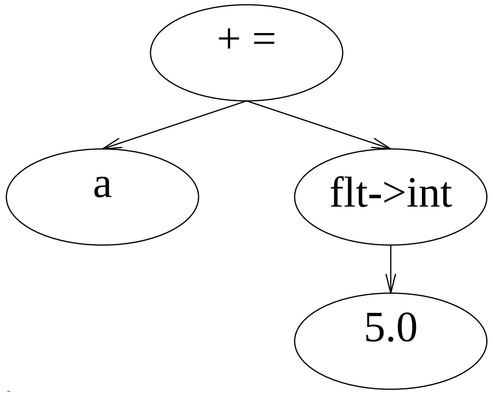


Semantic analysis: more analysis and type checking:

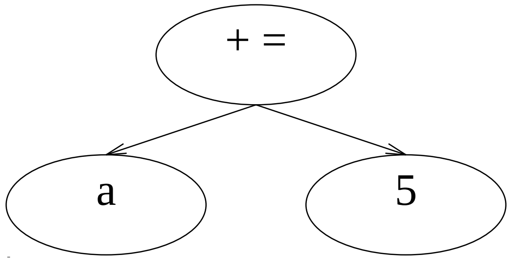


Structure of a compiler

Middle End



The middle end might eliminate the conversion, substituting the integer "5" for the float "5.0".



Code Generation

The code generator can significantly affect performance. There are many ways to compute "a+=5", some less efficient than others:

```
while $(t \neq a + 5)$ do
    $t \leftarrow rand()$
od
$a \leftarrow t$
```

While optimization can occur throughout the translation process, machine-independent transformations are typically relegated to the middle-end, while instruction selection and other machine-specific activities are pushed into code generation.

Bootstrapping a compiler

Often, a compiler is written in it "itself". That is, a compiler for PASCAL may be written in PASCAL. How does this work?

Initial Compiler for L on Machine M

1. The compiler can be written in a small subset of L, even though the compiler translates the full language.
2. A throw-away version of the subset language is implemented on M. Call this compiler α.
3. The L compiler can be compiled using the subset compiler, to generate a full compiler β.
4. The L compiler can also compile itself. The resulting object γ can be compared with β for verification.

Porting the Compiler

1. On machine M, the code generator for the full compiler is changed to target machine N.
2. Any program in L can now be cross-compiled from M to N.
3. The compiler can also be cross-compiled to produce an instance of γ that runs on machine N.

If the run-time library is mostly written in L, or in an intermediate language of β, then these can also be translated for N using the cross-compiler.

What else does a compiler do?

```
if (p)
   a = b + (c
else {d = f;
q = r;
```

Error detection. Strict language rules, consistently enforced by a compiler, increase the likelihood that a compiler-approved source program is bug-free.

Error diagnosis. Compilers can often assist the program author in addressing errors.

Error repair. Some ambitious compilers go so far as to insert or delete text to render the program executable.

```
for (i=1; i<=n; ++i)
{
    a[i] = b[i] + c[i]
}
```

Program optimization. The target produced by a compiler must be "observably equivalent" to the source interpretation. An optimizing compiler attempts to minimize the resource constraints (typically time and space) required by the target program.

Program instrumentation. The target program can be augmented with instructions and data to provide information for run-time debugging and performance analysis. Language features not checkable at compile-time are often checked at run-time by code inserted by the compiler.

Sophisticated error repair may include symbol insertion, deletion, and use of indentation structure.

Program optimization can significantly decrease the time spent on array index arithmetic. Since subscript ranges cannot in general be checked at compile-time, run-time tests may be inserted by the compiler.

Compiler design points – aquatic analogies

Powerboat Turbo–?. These compilers are fast, load-and-go. They perform little optimization, but typically offer good diagnostics and a good programming environment (sporting a good debugger). These compilers are well-suited for small development tasks, including small student projects.

Sailboat BCPL, Postscript. These compilers can do neat tricks but they require skill in their use. The compilers themselves are often small and simple, and therefore easily ported. They can assist in bootstrapping larger systems.

Tugboat C++ preprocessor, RATFOR. These compilers are actually front-ends for other (typically larger) back-ends. The early implementations of C++ were via a preprocessor.

Barge Industrial-strength. These compilers are developed and maintained with a company's reputation on the line. Commercial systems use these compilers because of their integrity and the commitment of their sponsoring companies to address problems. Increasingly these kinds of compilers are built by specialty houses such as Rational, KAI, etc.

Ferry Gnu compilers. These compilers are available via a General Public License from the Free Software Foundation. They are high-quality systems and can be built upon without restriction.

Another important design issue is the extent to which a compiler can respond *incrementally* to changes.

Compilers are taking over the world!

While compilers most prevalently participate in the translation of programming languages, some form of compiler technology appears in many systems:

Text processing Consider the "$\star$-roff" text processing pipe:

$$\text{PIC} \rightarrow \text{TBL} \rightarrow \text{EQN} \rightarrow \text{TROFF}$$

or the LaTeX pipe:

$$\text{\LaTeX} \rightarrow \TeX$$

each of which may produce

$$\text{DVI} \rightarrow \text{POSTSCRIPT}$$

Silicon compilers Such systems accept circuit specifications and compile these into VLSI layouts. The compilers can enforce the appropriate "rules" for valid circuit design, and circuit libraries can be referenced like modules in software library.

Compiler design *vs.* programming language design

Programming languages have	So compilers offer
Non-locals	Displays, static links
Recursion	Dynamic links
Dynamic Storage	Garbage collection
Call-by-name	Thunks
Modular structure	Interprocedural analysis
Dynamic typing	Static type analysis

It's expensive for a compiler to offer	So some languages avoid that feature
Non-locals	C
Call-by-name	C, PASCAL
Recursion	FORTRAN 66
Garbage collection	C

In general, *simple* languages such as C, PASCAL, and SCHEME have been more successful than complicated languages like PL/1 and ADA.

Language design for humans

```
Procedure $foo(x, y)$
    declare
        $x, y$   integer
        $a, b$   integer
        $\star p$    integer

    $p \leftarrow rand()$ ? &$a$ : &$b$
    $\star p \leftarrow x + y$

end
```

Syntactic simplicity. Syntactic signposts are kept to a minimum, except where aesthetics dictate otherwise: parentheses in C, semicolons in PASCAL.

Resemblance to mathematics. Infix notation, function names.

Flexible internal structures. Nobody would use a language in which one had to predeclare how many variables their program needed.

Freedom from specifying side-effects. What happens when p is dereferenced?

Programming language design is often a compromise between ease of use for humans, efficiency of translation, and efficiency of target code execution.

Language design for machines

```
(SymbolTable
   (NumSymbols 5)
   (Symbol
      (SymbolName x)
      (SymbolID   1)
   )
   (Symbol
      (SymbolName y)
      (SymbolID   2)
   )
   ...
)
(AliasRelations
   (NumAliasRelations 1)
   (AliasRelation
      (AliasID 1)
      (MayAliases 2 a b)
   )
)
```

```
(NodeSemantics
  (NodeID 2)
  (Def
    (DefID 2)
    (SymbID ?)
    (AliasWith 1)
    (DefValue
      (+
        (Use
          (UseID 1)
          (SymbID x)
        )
        (Use
          (UseID 2)
          (SymbID y)
        )
      )
    )
  )
)
```

We can require much more of our intermediate languages, in terms of details and syntactic form.

Compilers and target instruction sets

How should we translate $X = Y + Z$

In the course of its code generation, a simple compiler may use only 20% of a machine's potential instructions, because anomalies in an instruction set are difficult to "fit" into a code generator.

Consider two instructions

ADDREG R_1 R_2	$R_1 \leftarrow R_1 + R_2$
ADDMEM R_1 Loc	$R_1 \leftarrow R_1 + \star Loc$

Each instruction is *destructive* in its first argument, so Y and Z would have to be refetched if needed.

```
LOAD    1 Y
ADDMEM  1 Z
STORE   1 X
```

A simpler model would be to do all arithmetic in registers, assuming a *nondestructive* instruction set, with a reserved register for results (say, R_0):

```
LOAD     1 Y
LOAD     2 Z
LOADREG  0 1
ADDREG   0 2
STORE    0 X
```

This code preserves the value of Y and Z in their respective registers.

A popular approach is to generate code assuming the nondestructive paradigm, and then use an instruction selector to optimize the code, perhaps using destructive operations.

Current wisdom on compilers and architecture

Architects should design "orthogonal" RISC instruction sets, and let the optimizer make the best possible use of these instructions. Consider the program

$$\textbf{for } i \leftarrow 1 \textbf{ to } 10 \textbf{ do } X \leftarrow A[i]$$

where A is declared as a 10-element array ($1 \ldots 10$).

The VAX **has an instruction essentially of the form**

$$Index(A, i, low, high)$$

with semantics

```
if ($low \leq i \leq high$) then
    return $(A + 4 \times i)$
else
    return $(error)$
fi
```

Internally, this instruction requires two tests, one multiplication, and one addition.

However, notice that the loop does not violate the array bounds of A. Moreover, in moving from $A[i]$ to $A[i+1]$, the new address can be calculated by adding 4 to the old address.

While the use of an $Index$ instruction may seem attractive, better performance can be obtained by providing smaller, faster instructions to a compiler capable of optimizing their use.

A small example of language translation

$L(Add) =$

sums of two digits, expressed expressed in the usual (infix) notation

That's not very formal. What do we mean by this?

$$\{0 + 4, 3 + 7, \ldots\}$$

```
input(s)
case (s)
  of("0+0") return(OK)
  ...
  of("9+9") return(OK)
  default   return(BAD)
endcase
```

The program shown on the left *recognizes* **the** Add **language. Suppose we want to** *translate* **strings in** Add **into their sum, expressed base-4.**

```
input(s)
case (s)
  of("0+0") return("0")
  ...
  of("5+7") return("30")
  ...
  of("9+9") return("102")
  default   oops(BAD)
endcase
```

A *language* is a *set of strings*. With 100 possibilities, we could easily list all strings in this (small) language. This approach seems like lots of work, especially for languages with infinite numbers of strings, like C. We need a finite specification for such languages.

Grammars

The grammar below *generates* the Add language:

$$\begin{array}{lcl} \textbf{S} & \rightarrow & \textbf{D + D} \\ \textbf{D} & \rightarrow & \textbf{0} \\ & | & \textbf{1} \\ & | & \textbf{2} \\ & & \vdots \\ & | & \textbf{9} \end{array}$$

A grammar is formally

$$G = (V, \Sigma, P, S)$$

where

V **is the set of nonterminals. These appear on the left side of rules.**

Σ **is an alphabet of terminal symbols, that cannot be rewritten.**

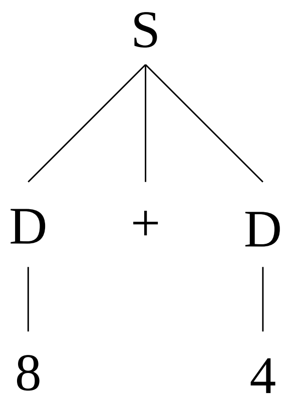
P **is a set of rewrite rules.**

S **is the *start* or *goal* symbol.**

The process by which a terminal string is created is called a *derivation*.

$$\begin{array}{lcl} \textbf{S} & \Rightarrow & \textbf{D + D} \\ & \Rightarrow & \textbf{8 + D} \\ & \Rightarrow & \textbf{8 + 4} \end{array}$$

This is a *leftmost* derivation, since a string of nonterminals is rewritten from the left. A tree illustrates how the grammar and the derivation *structure* the string:



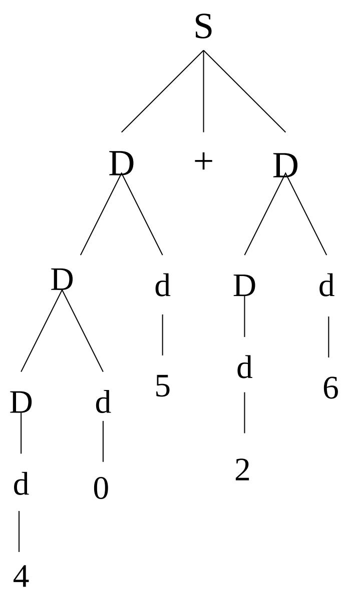
The above could be called a derivation tree, a (concrete) syntax tree, or a parse tree.

Strings in $L(G)$ are constructed by rewriting the symbol S according to the rules of P until a terminal string is derived.

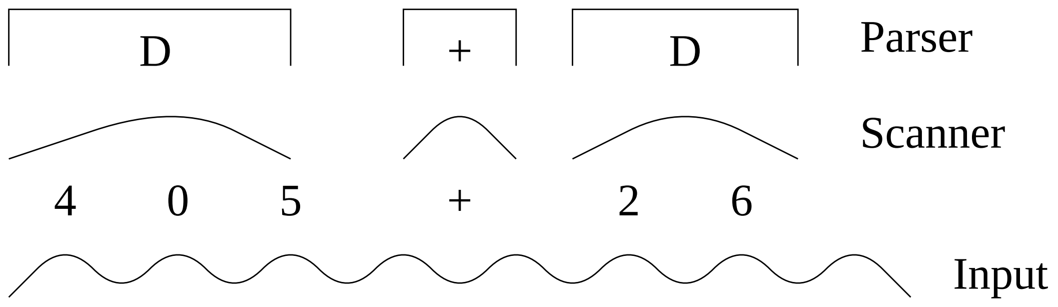
Sums of two numbers

Consider the set of strings that represent the sum of two numbers, such as $405 + 26$**. We could rewrite the grammar, as shown below:**

$$
\begin{array}{lcl}
\mathbf{S} & \rightarrow & \mathbf{D + D} \\
\mathbf{D} & \rightarrow & \mathbf{D\ d} \\
 & | & \mathbf{d} \\
\mathbf{d} & \rightarrow & \mathbf{0} \\
 & | & \mathbf{1} \\
 & \vdots & \\
 & | & \mathbf{9}
\end{array}
$$



Another solution would be to have a separate *tokenizing* process feed "D"s to the grammar, so that the grammar remains unchanged.



Scanners

Scanners are often the ugliest part of a compiler, but when cleverly designed, they can greatly simplify the design and implementation of a parser.

Typical tasks for a scanner:

- **Recognize reserved keywords.**
- **Find integer and floating-point constants.**
- **Ignore comments.**
- **Treat blank space appropriately.**
- **Find string and character constants.**
- **Find identifiers (variables).**

The C **statement**

```
if (++x==5) foo(3);
```

might be tokenized as

$\boxed{\textbf{if}}\ \boxed{\textbf{(}}\ \boxed{++}\ \boxed{\textbf{ID}}\ \boxed{==}\ \boxed{\textbf{5}}\ \boxed{\textbf{)}}\ \boxed{\textbf{ID}}\ \boxed{\textbf{(}}\ \boxed{\textbf{int}}\ \boxed{\textbf{)}}$

Unusual tasks for a scanner:

- **In (older)** FORTRAN, **blanks are optional. Thus, the phrases**

  `DO10I=1,5` **and** `DO10I=1.5`

  are distinguished only by the comma vs. the decimal. The first statement is the start of a DO loop, while the second statement assigns the variable `DO10I`**.**
- **In** C, **variables can be declared by built-in or by user-defined types. Thus, in**

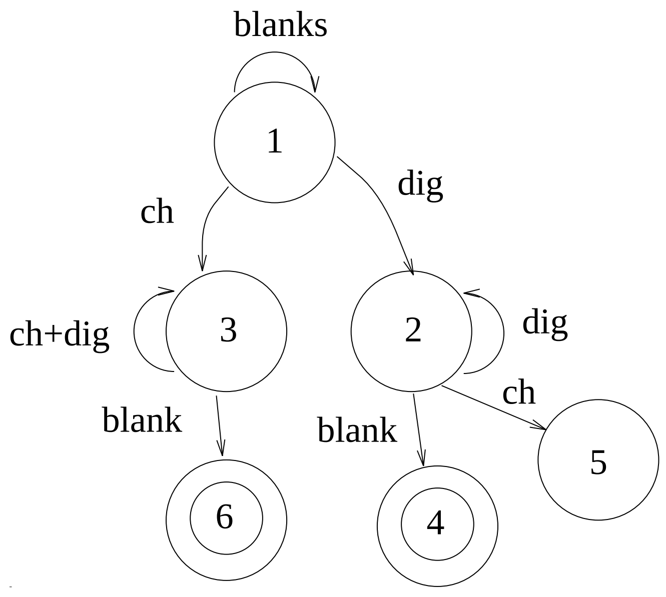
  ```
  foo x,y;
  ```

  the C **grammar needs to know that** `foo` **is a type name, and not a variable name.**

The balance of work between scanner and parser is typically dictated by restrictions of the parsing method and by a desire to make the grammar as simple as possible.

Scanners and Regular Languages

Most scanners are based on a simple computational model called the *finite-state automaton*.



These machines recognize *regular languages*.

To implement a finite-state transducer one begins with a GOTO **table that defines transitions between states:**

	GOTO **table**		
State	**ch**	**dig**	**blank**
1	**3**	**2**	**1**
2	**5**	**2**	**4**
3	**3**	**3**	**6**
4	**3**	**2**	**4**
5	**5**	**5**	**5**
6	**3**	**2**	**6**

which is processed by the driver

```
$state \leftarrow 1$
while (true) do
    $c \leftarrow NextSym()$
    /⋆ Do action ACTION[$state$][$c$] ⋆/
    $state \leftarrow$ GOTO[$state$][$c$]
od
```

Notice the similarity between states 1, 4, and 6.

Transduction

While the finite-state mechanism *recognizes* **appropriate strings, action must now be taken to construct and supply tokens to the parser. Between states, actions are performed as prescribed by the** ACTION **table shown below.**

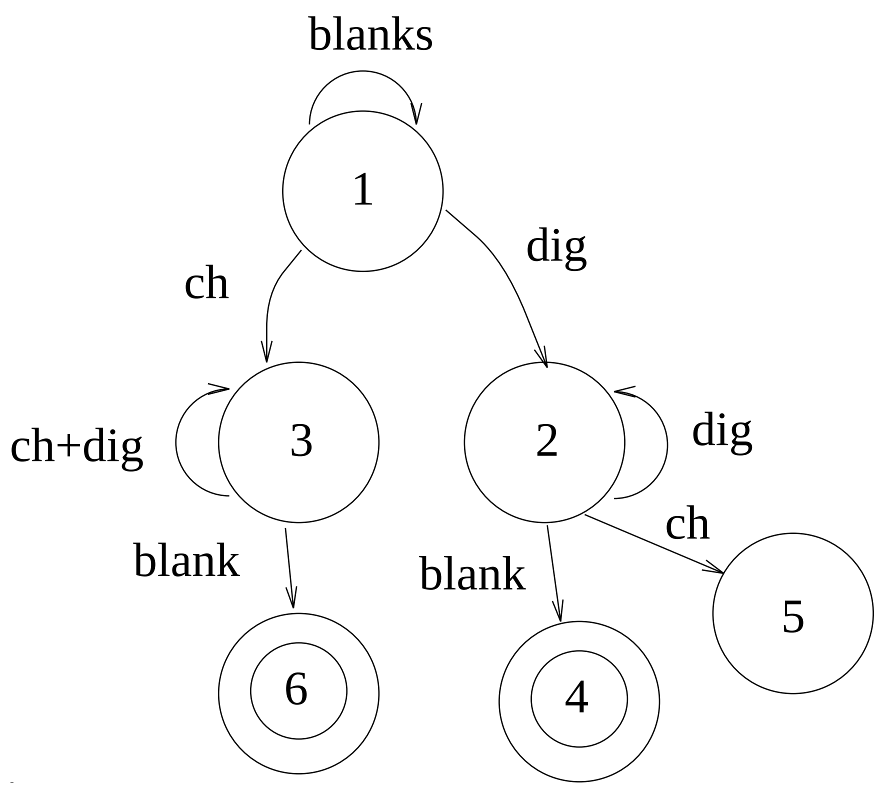
	ACTION **table**		
State	**ch**	**dig**	**blank**
1	**1**	**2**	**3**
2	**4**	**5**	**6**
3	**7**	**7**	**8**
4	**1**	**2**	**3**
5	**4**	**4**	**4**
6	**1**	**2**	**3**

Actions

1. $ID = ch$
2. $Num = dig$
3. **Do nothing**
4. **Error**
5. $Num = 10 \times Num + dig$
6. **return NUM**
7. $ID = ID || ch$
8. **return ID**

Technically, the ability to perform arbitrary actions makes our tokenizer more powerful than a finite-state automaton. Nonetheless, the underlying mechanism is quite simple, and can in fact be automatically generated. . . .

Regular grammars



$$
\begin{array}{lll}
\boxed{1} & \rightarrow & \textbf{blank}\ \boxed{1} \\
 & | & \textbf{ch}\ \boxed{3} \\
 & | & \textbf{dig}\ \boxed{2} \\
\boxed{2} & \rightarrow & \textbf{dig}\ \boxed{2} \\
 & | & \textbf{ch}\ \boxed{5} \\
 & | & \textbf{blank}\ \boxed{4} \\
\boxed{3} & \rightarrow & \textbf{ch}\ \boxed{3} \\
 & | & \textbf{dig}\ \boxed{3} \\
 & | & \textbf{blank}\ \boxed{6} \\
\boxed{4} & \rightarrow & \lambda \\
\boxed{6} & \rightarrow & \lambda
\end{array}
$$

In a regular grammar, each rule is of the form

$$
\begin{array}{lll}
A & \rightarrow & a\ A \\
A & \rightarrow & a
\end{array}
$$

where $A \in V$ and $a \in (\Sigma \cup \{\,\lambda\,\})$.

LEX as a scanner

First, define character classes:

```
ucase   [A-Z]
lcase   [a-z]
letter  ({ucase}|{lcase})
zero    0
nonzero [1-9]
sign    [+-]
digit   ({zero}|{nonzero})
blanks  [ \t\f]
newline \n
```

Next, specify patterns and actions:

`{L}({L}|{D})*`	`{ String(yytext); return(ID); }`
`‘‘++’’`	`{ return(IncOP); }`

In selecting which pattern to apply, LEX uses the following rules:

1. LEX **always tries for the longest match. If any pattern can “keep going” then** LEX **will keep consuming input until that pattern finishes or “gives up”. This property frequently results in buffer overflow for improperly specified patterns.**
2. LEX **will choose the pattern and action that succeeds by consuming the most input.**
3. **If there are ties, then the pattern specified earliest to** LEX **wins.**

The notation used above is *regular expression* notation, which allows for choice, catenation, and repeats. One can show by construction that any language accepted by a finite-state automaton has an equivalent regular expression.

A comment

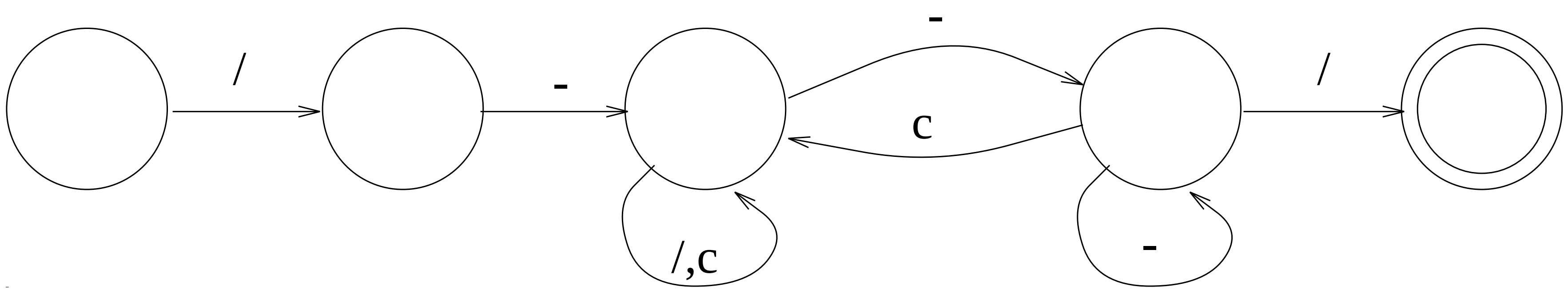
An interesting example is the C-like *comment* specification, which might be tempting to specify as:

```
"/-"  .*  "-/"
```

But in a longest match, this pattern will match the beginning of the first comment to the end of the last comment, and everything in between. If LEX's buffers don't overflow, most of the input program will be ignored by this faulty specification.

A better specification can be determined as follows:

1. Start with the wrong specification.
2. Construct the associated deterministic FSA.
3. Edit the FSA to cause acceptance at the end of the first comment (shown below).
4. Construct the regular expression associated with the resulting FSA.



with the corresponding regular expression

```
/- [ (/|c)* -(-)* c ]* (/|c)* -(-)* /
```

Teaching regular languages and scanners

Classroom

1. Motivate the study with examples from programming languages and puzzles (THINK-A-DOT, etc.).
2. Present deterministic FSA (DFA).
3. Present nondeterministic FSA (NFA).
4. Show how to construct NFAs from regular expressions.
5. Show good use of the empty string (λ or ϵ).
6. Eliminate the empty string.
7. Eliminate nondeterminism.
8. Minimize any DFA.
9. Construction of regular expressions from DFA.
10. Show the correspondence between regular grammars and FSAs.
11. The pumping lemma and nonregular languages.

Projects and Homework

1. Implement THINK-A-DOT.
2. Check if a YACC grammar is regular. If so, then emit the GOTO table for a finite-state driver.
3. Augment the above with ACTION clauses.
4. Process a YACC file for reserved keyword specifications:

   ```
   %token <rk> then
   ```

   and generate the appropriate pattern and action for recognizing these:

   ```
   "then" { return(THEN); }
   ```
5. Show that regular expression notation is itself not regular.

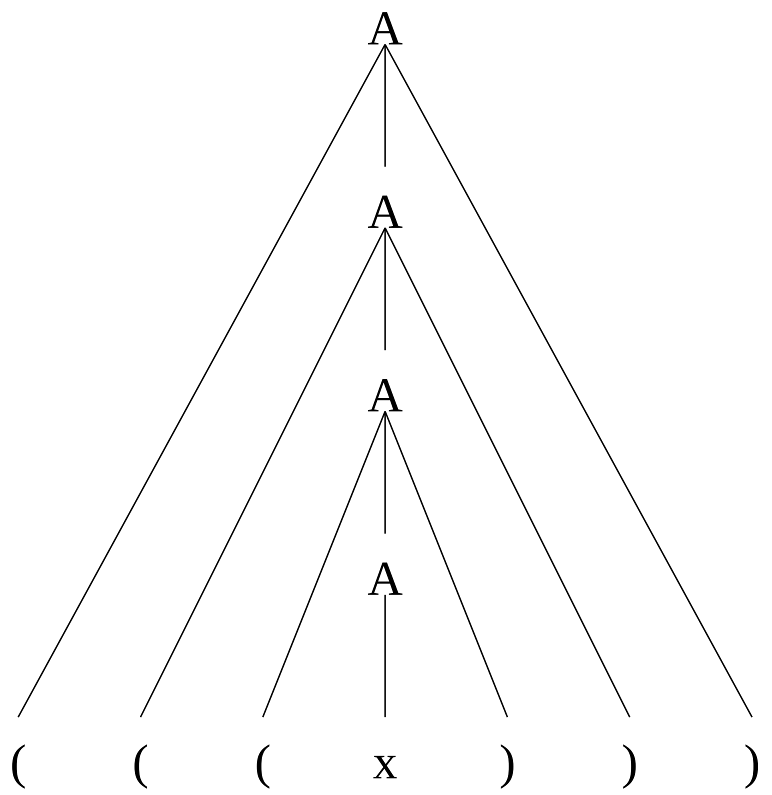
Some useful resources: (24, 28, 16, 2, 26).

Nonregular languages

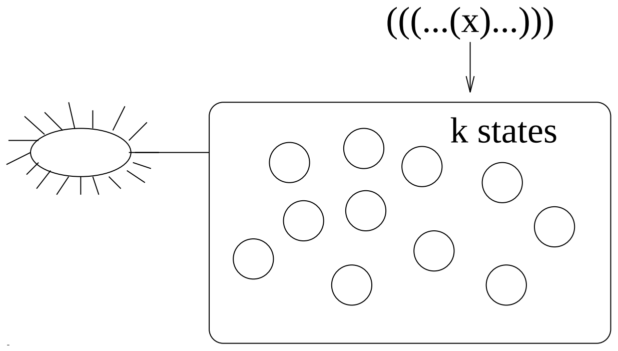
To grow beyond regular languages, we now allow a rule's right-hand side to contain any string of terminals or nonterminals.

$$\begin{array}{lcl} \textbf{A} & \rightarrow & \textbf{(A)} \\ & | & \textbf{x} \end{array}$$

describes the language $(^n x)^n$.



Suppose that some finite-state machine M of k states can recognize $\{\, (^n x)^n \,\}$.



Consider the input string $z = (^k x)^k$. After processing the k^{th} '(', some state must have been visited twice. By repeating the portion of z causing this loop, we obtain a string

$$(^k (^j x)^k, k \geq 0, j > 0$$

which is not in the language, but is accepted by M.

Since the proof did not depend on any particular k, we have shown that no finite-state machine can accept exactly this language.

Some more sums

Grammar

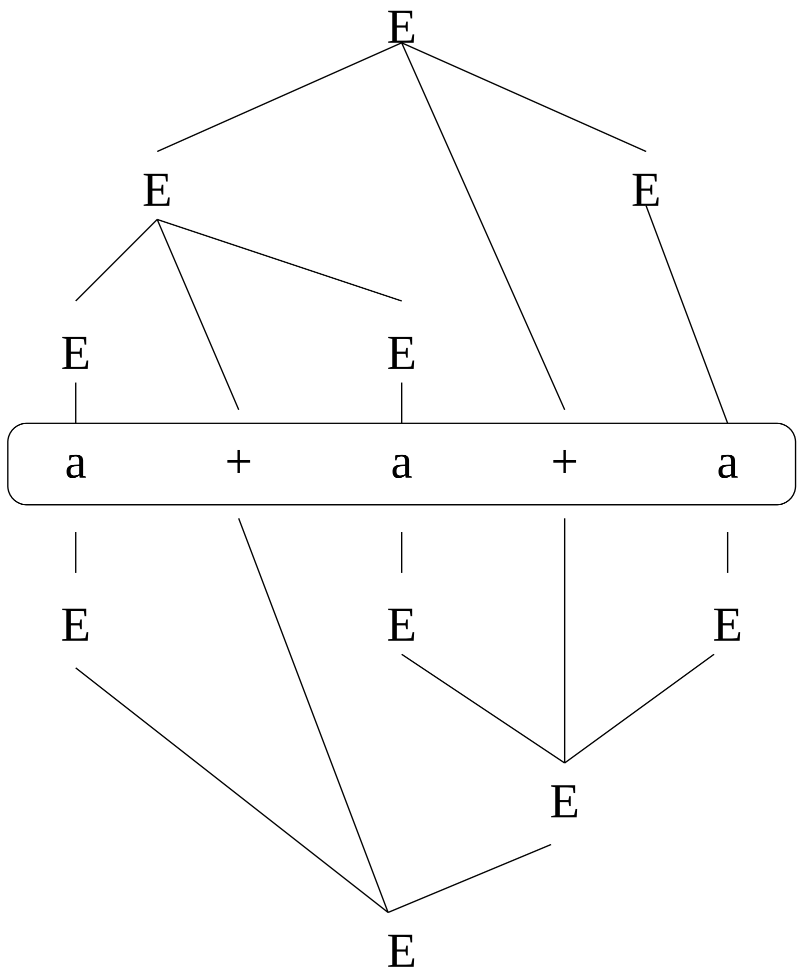
$$E \rightarrow E + E$$
$$\mid \; a$$

Leftmost derivation

$$\begin{aligned} E &\Rightarrow \boxed{E}+E \\ &\Rightarrow \boxed{E+E}+E \\ &\Rightarrow a+E+E \\ &\Rightarrow a+a+E \\ &\Rightarrow a+a+a \end{aligned}$$

Another leftmost derivation

$$\begin{aligned} E &\Rightarrow \boxed{E}+E \\ &\Rightarrow \boxed{a}+E \\ &\Rightarrow a+E+E \\ &\Rightarrow a+a+E \\ &\Rightarrow a+a+a \end{aligned}$$

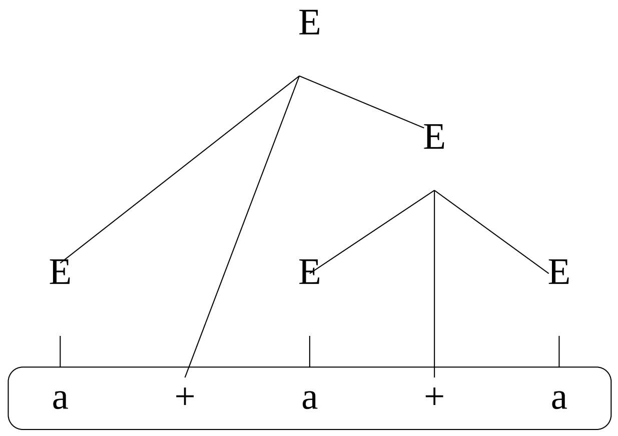


If the same string has two parse trees by a grammar G, then G is *ambiguous*. Equivalently, there are two distinct leftmost derivations of some string. Note that the language above is *regular*.

Ambiguity

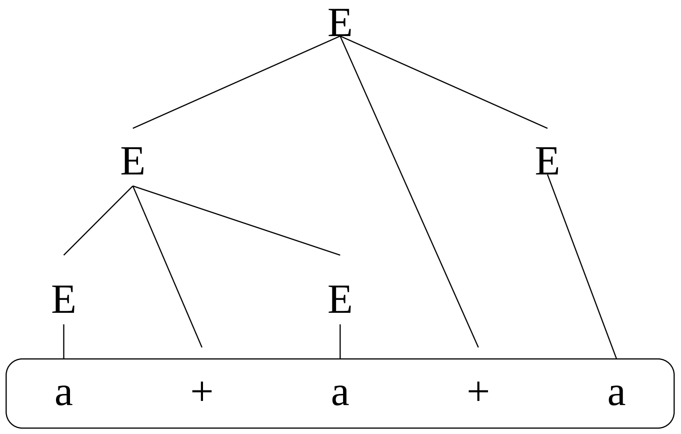
The parse tree below structures the input string as

$$(a + (a + a))$$



The parse tree below structures the input string as

$$((a + a) + a)$$



- With addition, the two expressions may be semantically the same. What if the a's were the operands of subtraction?
- How could a compiler choose between multiple parse trees for a given string?
- Unfortunately, there is (provably) no mechanical procedure for determining if a grammar is ambiguous; this is a job for human intelligence. However, compiler construction tools such as YACC can greatly facilitate the location and resolution of grammar ambiguities.
- It's important to emphasize the difference between a *grammar* being ambiguous, and a *language* being (inherently) ambiguous. In the former case, a different grammar may resolve the ambiguity; in the latter case, there exists no unambiguous grammar for the language.

Syntactic ambiguity

A great source of humor in the English language arises from our ability to construct interesting syntactically ambiguous phrases:

1. **I fed the elephant in my tennis shoes.**
   What does "in my tennis shoes" modify?
   (a) Was I wearing my tennis shoes while feeding the elephant?
   (b) Was the elephant wearing or inside my tennis shoes?
2. **The purple people eater.** What is purple?
   (a) Is the eater purple?
   (b) Are the people purple?

Suppose we modified the grammar for C, so that any {...} block could be treated as a primary value.

```
{ int i; i=3*5; } + 27;
```

would seem to have the value 42. But if we just rearrange the white space, we can get

```
{int i; i=3*5; }
+27;
```

which represents two statements, the second of which begins with a unary plus.

A good assignment along these lines is to modify the C grammar to allow this simple language extension, and ask the students to determine what went wrong. The students should be fairly comfortable using YACC before trying this experiment.

Semantic ambiguity

In English, we can construct sentences that have only one parse, but still have two different meanings:

1. **Milk drinkers turn to powder.** Are more milk drinkers using powdered milk, or are milk drinkers rapidly dehydrating?
2. **I cannot recommend this student too highly.** Do words of praise escape me, or am I unable to offer my support.

In programming languages, the language standard must make the meaning of such phrases clear, often by applying elements of context.

For example, the expression

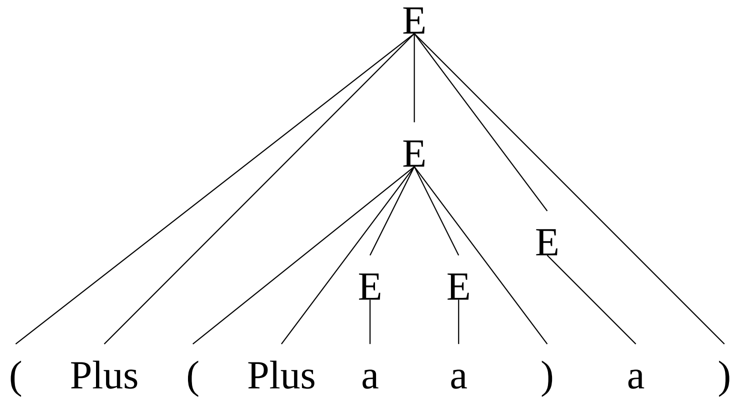
$$a + b$$

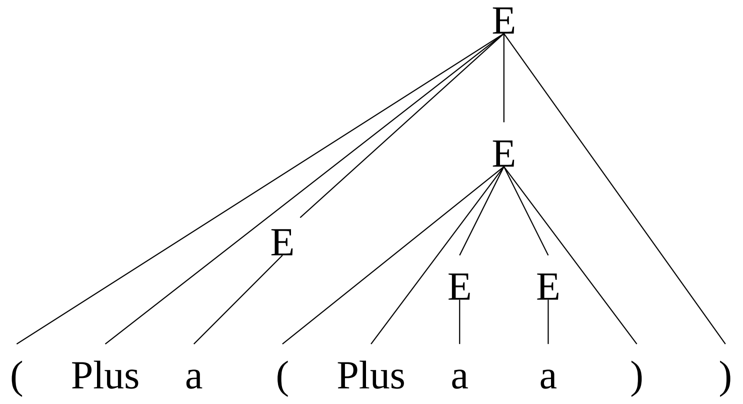
could connote an integer or floating-point sum, depending on the types of a and b.

A nonambiguous grammar

$$\begin{array}{lcl} \textbf{E} & \rightarrow & \textbf{(Plus E E)} \\ & | & \textbf{(Minus E E)} \\ & | & \textbf{a} \end{array}$$

It's interesting to note that the above grammar, intended to generate LISP**-like expressions, is not ambiguous.**





is the prefix equivalent of

$$((a + a) + a)$$

is the prefix equivalent of

$$(a + (a + a))$$

These are two *different strings* from this language, each associated explicitly with a particular grouping of the terms. Essentially, the parentheses are syntactic sentinels that simplify construction of an unambiguous grammar for this language.

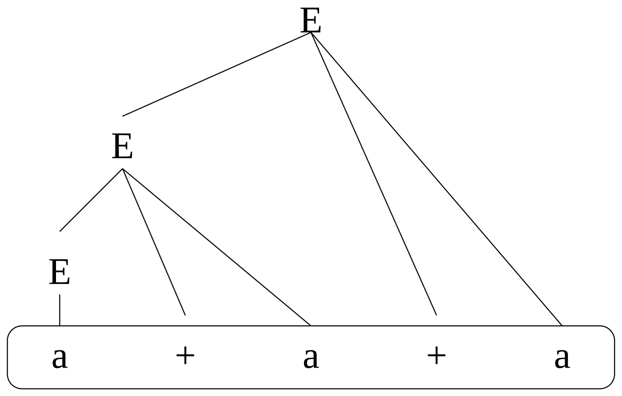
Addressing ambiguity

$$\begin{array}{lcl} E & \rightarrow & E + E \\ & \mid & a \end{array}$$

We'll try to rewrite the above grammar, so that in a (leftmost) derivation, there's only one rule choice that derives longer strings.

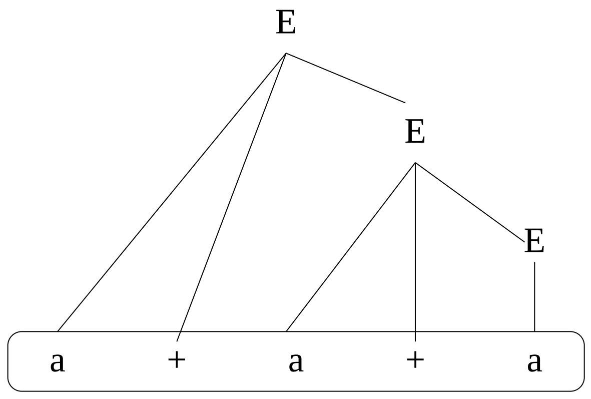
$$\begin{array}{lcl} E & \rightarrow & E + a \\ & \mid & E - a \\ & \mid & a \end{array}$$

These rules are *left recursive*, and the resulting derivations tend to associate operations from the left:



$$\begin{array}{lcl} E & \rightarrow & a + E \\ & \mid & a - E \\ & \mid & a \end{array}$$

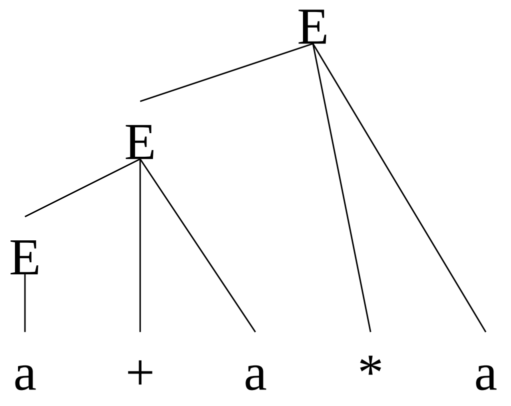
The grammar is still unambiguous, but strings are now associated from the right:



Addressing ambiguity (cont'd)

Our first try to expand our grammar might be:

$$\begin{array}{lcl} \textbf{E} & \rightarrow & \textbf{E + a} \\ & | & \textbf{E} * \textbf{a} \\ & | & \textbf{a} \end{array}$$



The above parse tree does not reflect the usual precedence of $*$ over $+$.

To obtain *sums of products*, we revise our grammar:

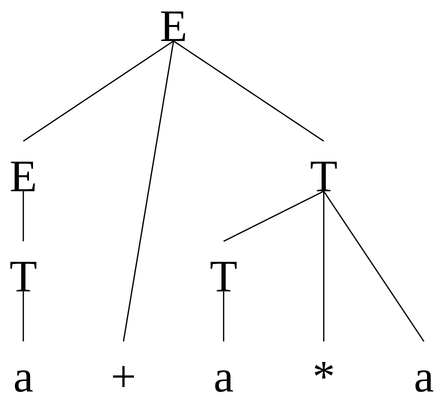
$$\begin{array}{lcl} \textbf{E} & \rightarrow & \textbf{E + T} \\ & | & \textbf{T} \end{array}$$

This generates strings of the form

$$T + T + \ldots + T$$

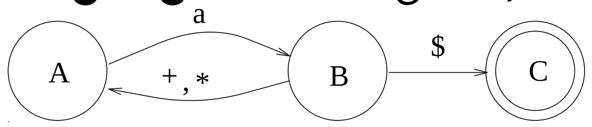
We now allow each T to generate strings of the form $a * a * \ldots * a$

$$\begin{array}{lcl} \textbf{E} & \rightarrow & \textbf{E + T} \\ & | & \textbf{T} \\ \textbf{T} & \rightarrow & \textbf{T} * \textbf{a} \\ & | & \textbf{a} \end{array}$$

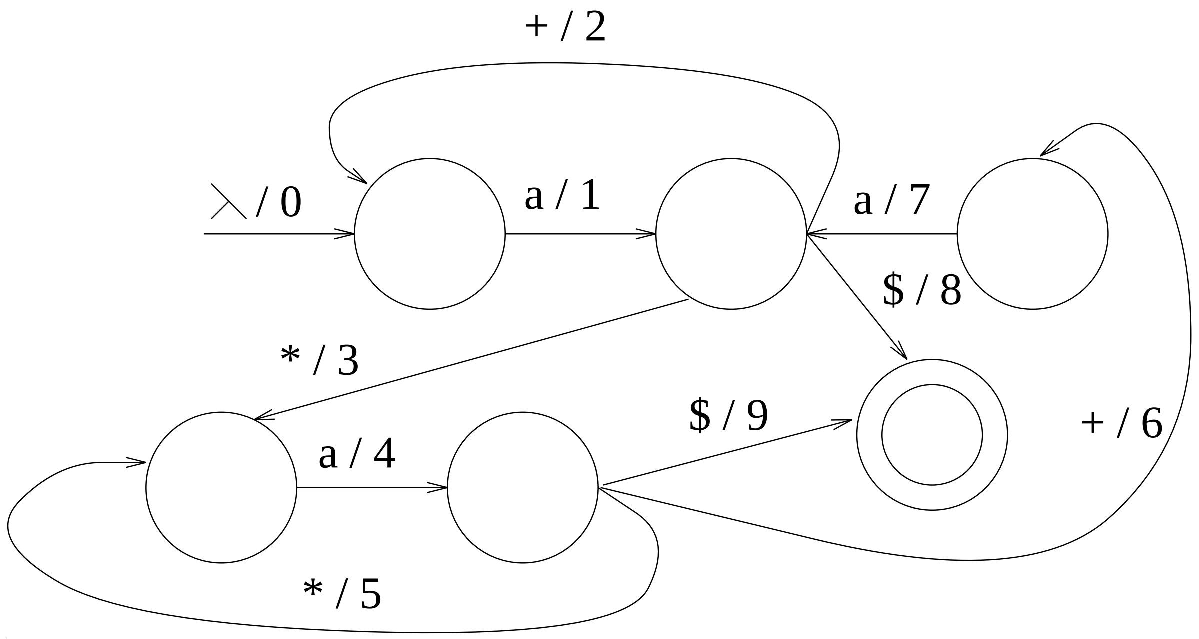


Translating two-level expressions

Since our language is still *regular*, a finite-state machine could do the job. While the machine



could do the job, there's not enough "structure" to this machine to accomplish the prioritization of $*$ over $+$. However, the machine below can do the job.



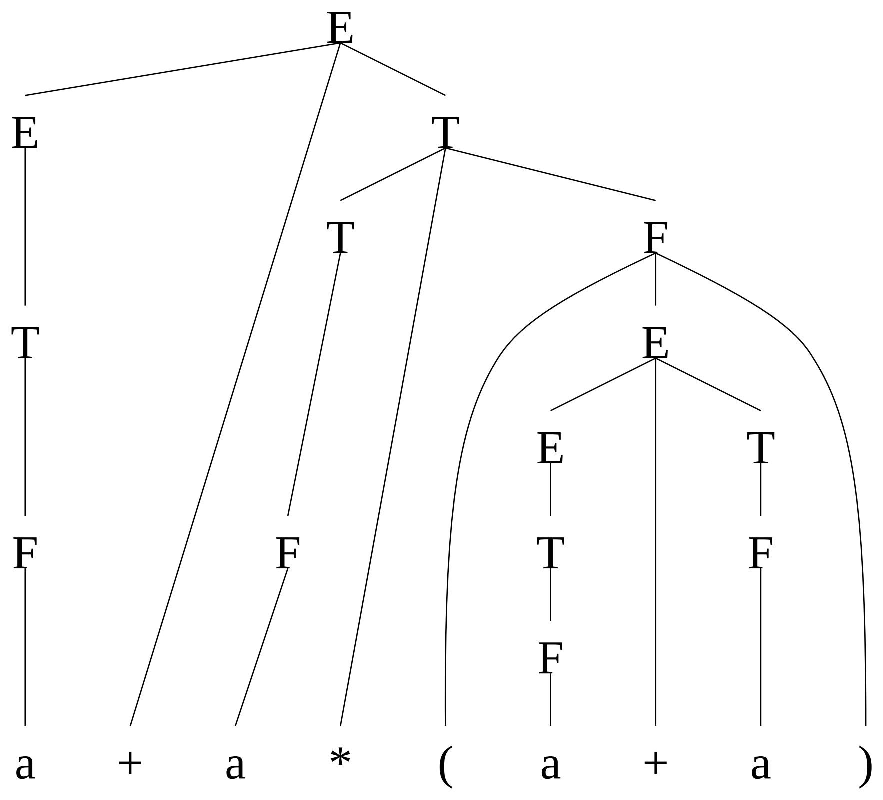
0	$Sum = 0$	5	$Prod = Prod \times Acc$
1	$Acc = a$	6	$Sum = Sum + (Prod \times Acc);\ Prod = 1$
2	$Sum = Sum + Acc$	7	$Acc = a$
3	$Prod = Prod \times Acc$	8	$Sum = Sum + Acc$
4	$Acc = a$	9	$Sum = Sum + (Prod \times Acc);\ Prod = 1$

Let's add parentheses

While our grammar currently structures inputs appropriately for operator priorities, parentheses are typically introduced to override default precedence. Since we want a parenthesized expression to be treated "atomically", we now generate sums of products of parenthesized expressions.

$$
\begin{array}{lcl}
\mathbf{E} & \rightarrow & \mathbf{E + T} \\
 & | & \mathbf{T} \\
\mathbf{T} & \rightarrow & \mathbf{T} * \mathbf{F} \\
 & | & \mathbf{F} \\
\mathbf{F} & \rightarrow & \mathbf{(E)} \\
 & | & \mathbf{a}
\end{array}
$$

This grammar generates a nonregular language. Therefore, we need a more sophisticated "machine" to parse and translate its generated strings.



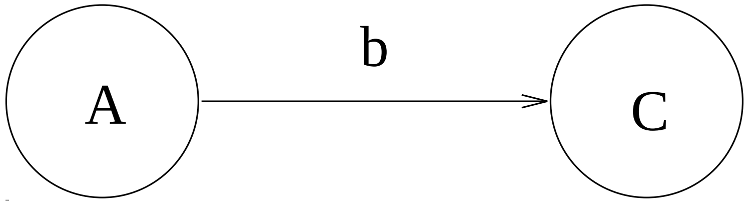
The grammar we have developed thus far is the textbook "expression grammar". Of course, we should make a into a nonterminal that can generate identifiers, constants, procedure calls, *etc*.

Beyond finite-state machines

For a rule of the form

$$\mathbf{A} \rightarrow \mathbf{b\,C}$$

we developed a finite-state mechanism of the form



After arrival at C, there is no need to remember how we got there.

Now, with a rule such as

$$\mathbf{F} \rightarrow \mathbf{(\,E\,)}$$

we cannot just arrive at an E and forget that we need exactly one closing parenthesis for each opening one that got us there.

Instead of "going to" a state E based on consuming an opening parenthesis, suppose we called a procedure E to consume all input ultimately derived from the nonterminal:

```
Procedure $F()$
    call $Expect(OpenParen)$
    call $E()$
    call $Expect(CloseParen)$
end
```

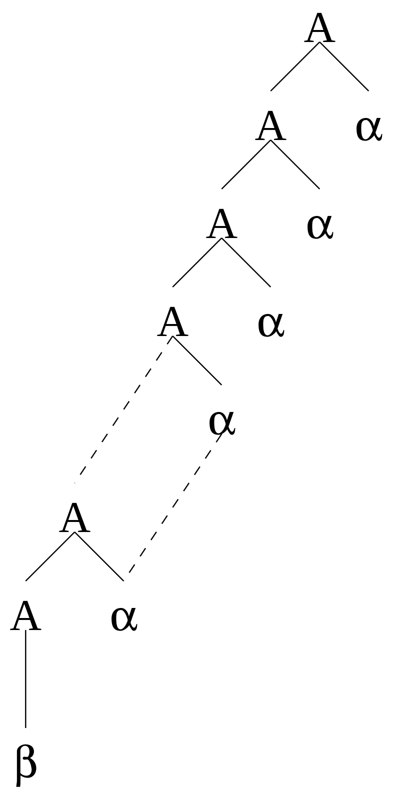
This style of parser construction is called *recursive descent*. **The procedure associated with each nonterminal is responsible for directing the parse through the right-hand side of the appropriate production.**

1. What about rules that are left-recursive?
2. What happens if there is more than one rule associated with a nonterminal?

Eliminating left recursion – grammar transformation

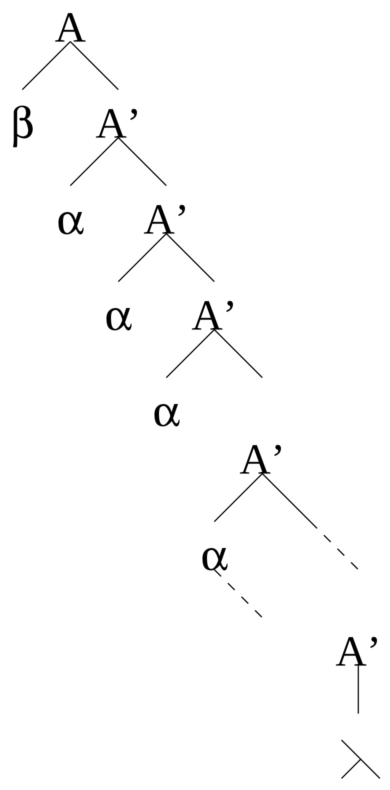
Original

$$\begin{aligned} \mathbf{A} &\rightarrow \mathbf{A}\,\alpha \\ &\mid\ \beta \end{aligned}$$



Transformed

$$\begin{aligned} \mathbf{A} &\rightarrow \beta\, A' \\ A' &\rightarrow \alpha\, A' \mid \lambda \end{aligned}$$

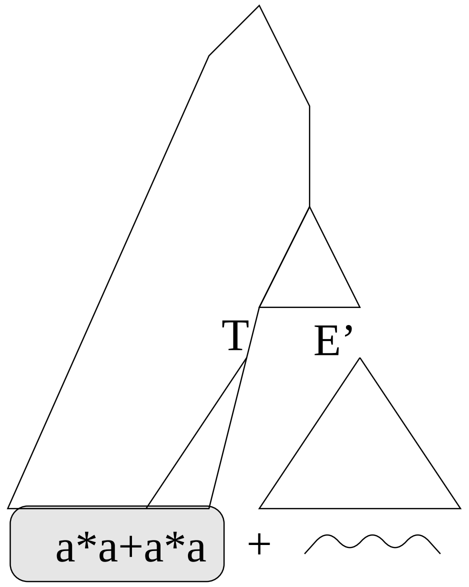


The two grammars generate the same language, but the one on the right generates the β first, and then a string of αs, using a rule that is *right* recursive instead of left recursive.

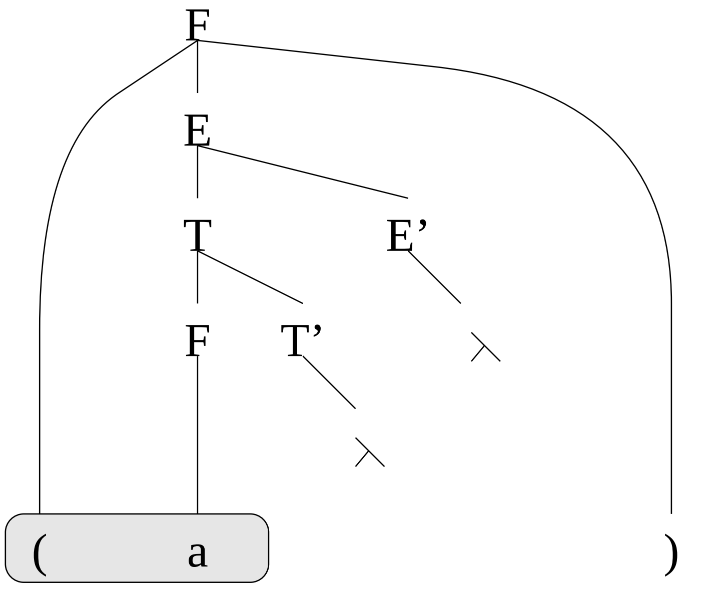
The transformed expression grammar

E	$\rightarrow$	**T** E'
E'	$\rightarrow$	**+ T** E'
E'	$\rightarrow$	**- T** E'
	$\mid$	λ
T	$\rightarrow$	**F** T'
T'	$\rightarrow$	$*$ **F** T'
T'	$\rightarrow$	**/ F** T'
	$\mid$	λ
F	$\rightarrow$	**(E)**
	$\mid$	**a**

Which rule to choose?



And what about λ?



First sets

$$First(\alpha) = \begin{cases} \{\,\alpha\,\} & \textbf{if } \alpha \in \Sigma \\ \cup_{(\alpha \to \omega_i) \in P} First(\omega_i) & \textbf{if } \alpha \in V \\ \{\,\lambda\,\} & \textbf{if } \alpha = \lambda \end{cases}$$

$$First(\alpha_1 \ldots \alpha_L) = \bigcup_{j \mid \forall_{k=1}^{j-1}(\lambda \in First(\alpha_k))} First(\alpha_j)$$

A	$\rightarrow$	**B C**
	$\mid$	**E F G H**
	$\mid$	**H**
B	$\rightarrow$	**b**
C	$\rightarrow$	λ
	$\mid$	**c**
E	$\rightarrow$	λ
	$\mid$	**e**
F	$\rightarrow$	**C E**
G	$\rightarrow$	**g**
H	$\rightarrow$	λ
	$\mid$	**h**

ω	**First(ω)**
H	$\{\,h, \lambda\,\}$
G	$\{\,g\,\}$
C	$\{\,c, \lambda\,\}$
B	$\{\,b\,\}$
E	$\{\,e, \lambda\,\}$
F	$\{\,c, e, \lambda\,\}$
A	$\{\,b, e, c, g, h, \lambda\,\}$
BC	$\{\,b\,\}$
EFGH	$\{\,e, c, g\,\}$

Follow sets

1. **Initially set** $Follow(N) = \emptyset, \forall\, N \in V$.
2. **Given production** $A \rightarrow \alpha B \beta$**, set**

$$Follow(B) = Follow(B) \cup (First(\beta) - \{\,\lambda\,\})$$

3. **Given production** $A \rightarrow \alpha B \beta$**, where** $\lambda \in First(\beta)$**, set**

$$Follow(B) = Follow(B) \cup Follow(A)$$

$$\begin{array}{lcl}
\textbf{A} & \rightarrow & \textbf{B C} \\
 & \mid & \textbf{E F G H} \\
 & \mid & \textbf{H} \\
\textbf{B} & \rightarrow & \textbf{b} \\
\textbf{C} & \rightarrow & \lambda \\
 & \mid & \textbf{c} \\
\textbf{E} & \rightarrow & \lambda \\
 & \mid & \textbf{e} \\
\textbf{F} & \rightarrow & \textbf{C E} \\
\textbf{G} & \rightarrow & \textbf{g} \\
\textbf{H} & \rightarrow & \lambda \\
 & \mid & \textbf{h}
\end{array}$$

N	**Follow(N)**		
A			$\{\,\}$
B	$First(C) \cup Follow(A)$	=	$\{\,c\,\}$
F	$First(G)$	=	$\{\,g\,\}$
C	$Follow(A) \cup First(E)$		
	$\cup\, Follow(F)$	=	$\{\,e, g\,\}$
E	$First(F) \cup First(G)$	=	$\{\,c, e, g\,\}$
G	$First(H) \cup Follow(A)$	=	$\{\,h\,\}$
H	$Follow(A)$	=	$\{\,\}$

Recursive descent parser generation

```
**Procedure** $NonTermN$
    **if** $(LookAhead() \in First(\omega_1),$ **where** $(N \to \omega_1) \in P)$ **then**
        /* Use $\omega_1$ to generate calls to Expect() and other nonterminals */
    **else**
        **if** $(LookAhead() \in Follow(N)$ **and** $(N \to \lambda) \in P)$ **then**
            **return** ()
        **else**
            /* error */
        **fi**
    **fi**
**end**
```

Recursive descent – Example

S	$\rightarrow$	**A C $**
C	$\rightarrow$	**c**
	$\mid$	λ
A	$\rightarrow$	**a B C d**
	$\mid$	**B Q**
	$\mid$	λ
B	$\rightarrow$	**b B**
	$\mid$	**d**
Q	$\rightarrow$	**q**

	First	**Follow**
S	$\{a, b, d, c, \$\}$	$\{\ \}$
A	$\{a, b, d, \lambda\}$	$\{c, \$\}$
B	$\{b, d\}$	$\{c, d, q\}$
C	$\{c, \lambda\}$	$\{d, \$\}$
Q	$\{q\}$	$\{c, \$\}$

The generated procedures

S	$\rightarrow$	A C $
C	$\rightarrow$	c
	$\vert$	λ
A	$\rightarrow$	a B C d
	$\vert$	B Q
	$\vert$	λ
B	$\rightarrow$	b B
	$\vert$	d
Q	$\rightarrow$	q

	First	Follow
S	$\{a, b, d, c, \$\}$	$\{\ \}$
A	$\{a, b, d, \lambda\}$	$\{c, \$\}$
B	$\{b, d\}$	$\{c, d, q\}$
C	$\{c, \lambda\}$	$\{d, \$\}$
Q	$\{q\}$	$\{c, \$\}$

```
Procedure $S()$
    if $(LookAhead() \in \{a, b, d, c, \$\})$ then
        call $A()$
        call $C()$
        call $Expect(\$)$
    else
        /* error */
    fi

end

Procedure $C()$
    if $(LookAhead() \in \{c\})$ then
        call $Expect(c)$
    else
        if $(Lookahead() \notin \{d, \$\})$ then
            /* error */
        fi
    fi

end
```

The generated procedures (cont'd)

S	$\rightarrow$	**A C $**
C	$\rightarrow$	**c**
	$\mid$	λ
A	$\rightarrow$	**a B C d**
	$\mid$	**B Q**
	$\mid$	λ
B	$\rightarrow$	**b B**
	$\mid$	**d**
Q	$\rightarrow$	**q**

	First	**Follow**
S	$\{a, b, d, c, \$\}$	$\{\ \}$
A	$\{a, b, d, \lambda\}$	$\{c, \$\}$
B	$\{b, d\}$	$\{c, d, q\}$
C	$\{c, \lambda\}$	$\{d, \$\}$
Q	$\{q\}$	$\{c, \$\}$

```
Procedure $A()$
    if $(LookAhead() \in \{a\})$ then
        call $Expect(a)$
        call $B()$
        call $C()$
        call $Expect(d)$
    else
        if $(LookAhead() \in \{b, d\})$ then
            call $B()$
            call $Q()$
        else
            if $(LookAhead() \in \{c, \$\})$ then
                return ()
            else
                /* error */
            fi
        fi
    fi
end
```

The generated procedures (cont'd)

$$\begin{array}{lcl}
\textbf{S} & \rightarrow & \textbf{A C \$} \\
\textbf{C} & \rightarrow & \textbf{c} \\
 & | & \lambda \\
\textbf{A} & \rightarrow & \textbf{a B C d} \\
 & | & \textbf{B Q} \\
 & | & \lambda \\
\hline
\textbf{B} & \rightarrow & \textbf{b B} \\
 & | & \textbf{d} \\
\textbf{Q} & \rightarrow & \textbf{q} \\
\hline
\end{array}$$

	First	**Follow**
S	$\{a, b, d, c, \$\}$	$\{\}$
A	$\{a, b, d, \lambda\}$	$\{c, \$\}$
B	$\{b, d\}$	$\{c, d, q\}$
C	$\{c, \lambda\}$	$\{d, \$\}$
Q	$\{q\}$	$\{c, \$\}$

```
Procedure $B()$
    if $(LookAhead() \in \{b\})$ then
        call $Expect(b)$
        call $B()$
    else
        if $(LookAhead() \in \{d\})$ then
            call $Expect(d)$
        else
            /* error */
        fi
    fi

end

Procedure $Q()$
    if $(LookAhead() \in \{q\})$ then
        call $Expect(q)$
    else
        /* error */
    fi

end
```

Recursive descent – expression grammar

E	$\rightarrow$	**T** E'
E'	$\rightarrow$	**+ T** E'
E'	$\rightarrow$	**- T** E'
	$\mid$	λ
T	$\rightarrow$	**F** T'
T'	$\rightarrow$	$*$ **F** T'
T'	$\rightarrow$	**/ F** T'
	$\mid$	λ
F	$\rightarrow$	**(E)**
	$\mid$	**a**

	First	**Follow**
E	$\{ (, a \}$	$\{), \$ \}$
E'	$\{ +, - \}$	$\{), \$ \}$
T	$\{ (, a \}$	$\{ +, -,), \$ \}$
T'	$\{ *, / \}$	$\{ +, -,), \$ \}$
F	$\{ (, a \}$	$\{ *, /, +, -,), \$ \}$

```
Procedure $E'$
    if $(LookAhead(+))$ then
        call $Expect(+)$
        call $T$
        call $E'$
    else
        if $(LookAhead(-))$ then
            call $Expect(-)$
            call $T$
            call $E'$
        else
            if $(LookAhead(\$, ')'))$ then
                return ()
            else
                call $Error()$
            fi
        fi
    fi
end
```

Maintaining lookahead

```
Procedure $main()$
    $LAtok \leftarrow GetNextToken()$
    call $S()$
end

Function $LookAhead()$ : $token$
    return $(LAtok)$
end

Procedure $Expect(tok)$
    if $(LAtok = tok)$ then
        $LAtok \leftarrow GetNextToken()$
    else
        /* error */
    fi
end
```

A lookahead of k tokens is maintained by appropriately buffering the input.

Technically, k lookahead is equivalent in power to a single token of lookahead. The proof is constructive: each permutation of k symbols is encoded as a single token.

The $Expect(tok)$ procedure first compares the incoming token against tok, and then advances input into the lookahead buffer.

Recursive descent – correctness and properties

When is our recursive descent parser construction successful? If the grammar involves any left-recursion, then our construction method will create a parser containing an infinite loop. So, we require that the grammar be free of left-recursion.

The grammar transformation technique covered earlier can help eliminate left-recursion.

Also, we require that the parser operate *deterministically*: **actions taken at each step make progress toward completion, so that backtracking is not necessary.**

Thus, given a set of rules for nonterminal N

$$\begin{array}{rcl} N & \rightarrow & \omega_1 \\ & | & \vdots \\ & | & \omega_n \end{array}$$

we require

1.

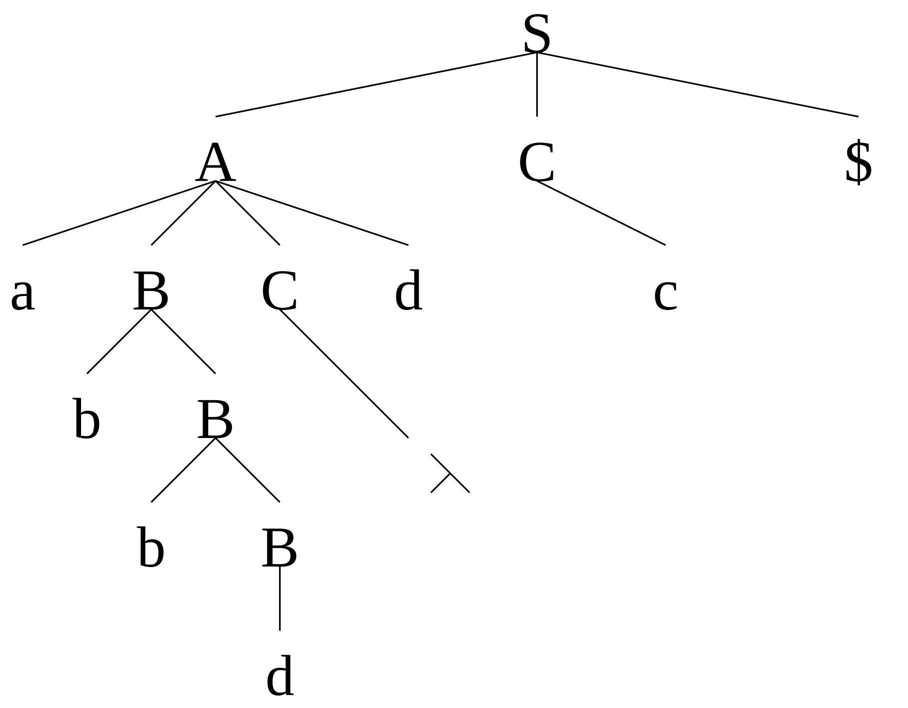
$$\bigcap_i First(\omega_i) = \{ \}$$

2. If $\lambda = \omega_j$, $1 \leq j \leq n$, then we also require

$$\bigcup_i (Follow(N) \cap First(\omega_i)) = \{ \}$$

Recursive descent and leftmost derivations

Let's examine how our recursive descent parser recognizes the string "abbddc$"



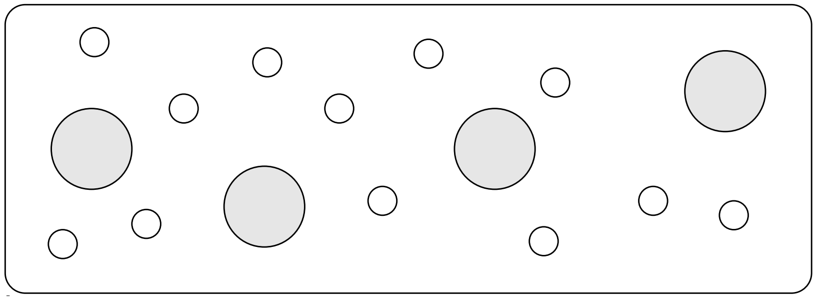
$\boxed{1}$	**S**	$\rightarrow$	**A C $**
$\boxed{2}$	**C**	$\rightarrow$	**c**
$\boxed{3}$		$\mid$	λ
$\boxed{4}$	**A**	$\rightarrow$	**a B C d**
$\boxed{5}$		$\mid$	**B Q**
$\boxed{6}$		$\mid$	λ
$\boxed{7}$	**B**	$\rightarrow$	**b B**
$\boxed{8}$		$\mid$	**d**
$\boxed{9}$	**Q**	$\rightarrow$	**q**

$$
\begin{aligned}
\mathbf{S} &\Rightarrow A\ \mathbf{C}\ \$ \\
&\Rightarrow \boxed{\mathbf{a}\ B\ \mathbf{C}\ \mathbf{d}}\ \mathbf{C}\ \$ \\
&\Rightarrow \mathbf{a}\ \boxed{\mathbf{b}\ B}\ \mathbf{C}\ \mathbf{d}\ \mathbf{C}\ \$ \\
&\Rightarrow \mathbf{a}\ \mathbf{b}\ \boxed{\mathbf{b}\ B}\ \mathbf{C}\ \mathbf{d}\ \mathbf{C}\ \$ \\
&\Rightarrow \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \boxed{\mathbf{d}}\ C\ \mathbf{d}\ \mathbf{C}\ \$ \\
&\Rightarrow \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \mathbf{d}\ \boxed{}\ \mathbf{d}\ C\ \$ \\
&\Rightarrow \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \mathbf{d}\ \mathbf{d}\ \boxed{\mathbf{c}}\ \$
\end{aligned}
$$

The procedure activations trace a left-most derivation of the string. We call this style of parsing *LL*, because it uses a *L*eft-most scan of the input and produces a *L*eft-most derivation.

In fact, the *record* of the parse is simply the order in which the grammar rules are applied: $\boxed{1}\ \boxed{4}\ \boxed{7}\ \boxed{7}\ \boxed{8}\ \boxed{3}\ \boxed{2}$

Error repair



Good programming languages are designed with a relatively large "distance" between syntactically correct programs, to increase the likelihood that conceptual mistakes are caught as syntactic errors.

Error repair usually occurs at two levels:

Local: repairs mistakes with little global import, such as missing semicolons and undeclared variables.

Scope: repairs the program text so that scopes are correct. Errors of this kind include unbalanced parentheses and begin/end blocks.

Repair actions can be divided into *insertions* and *deletions*. Typically the compiler will use some lookahead and backtracking in attempting to make progress in the parse. There is great variation among compilers, though some languages (PL/C) carry a tradition of good error repair. Goals of error repair include:

1. No input should cause the compiler to collapse.
2. Illegal constructs are flagged.
3. Frequently occurring errors are repaired gracefully.
4. Minimal stuttering or cascading of errors.

LL-style parsing lends itself well to error repair, since the compiler uses the grammar's rules to *predict* what should occur next in the input.

Augmenting recursive descent parsers for error recovery

Recursive and LL parsers are often called *predictive*, **because they operate by predicting the next step in a derivation.**

Suppose the parser is operating in procedure A **for some nonterminal** A. **If an error occurs, it seems reasonable to recover by skipping to a symbol that could follow** A, **and then return.**

E	$\rightarrow$	**T** E'
E'	$\rightarrow$	**+ T** E'
E'	$\rightarrow$	**- T** E'
	$\mid$	λ
T	$\rightarrow$	**F** T'
T'	$\rightarrow$	$*$ **F** T'
T'	$\rightarrow$	**/ F** T'
	$\mid$	λ
F	$\rightarrow$	**(E)**
	$\mid$	**a**

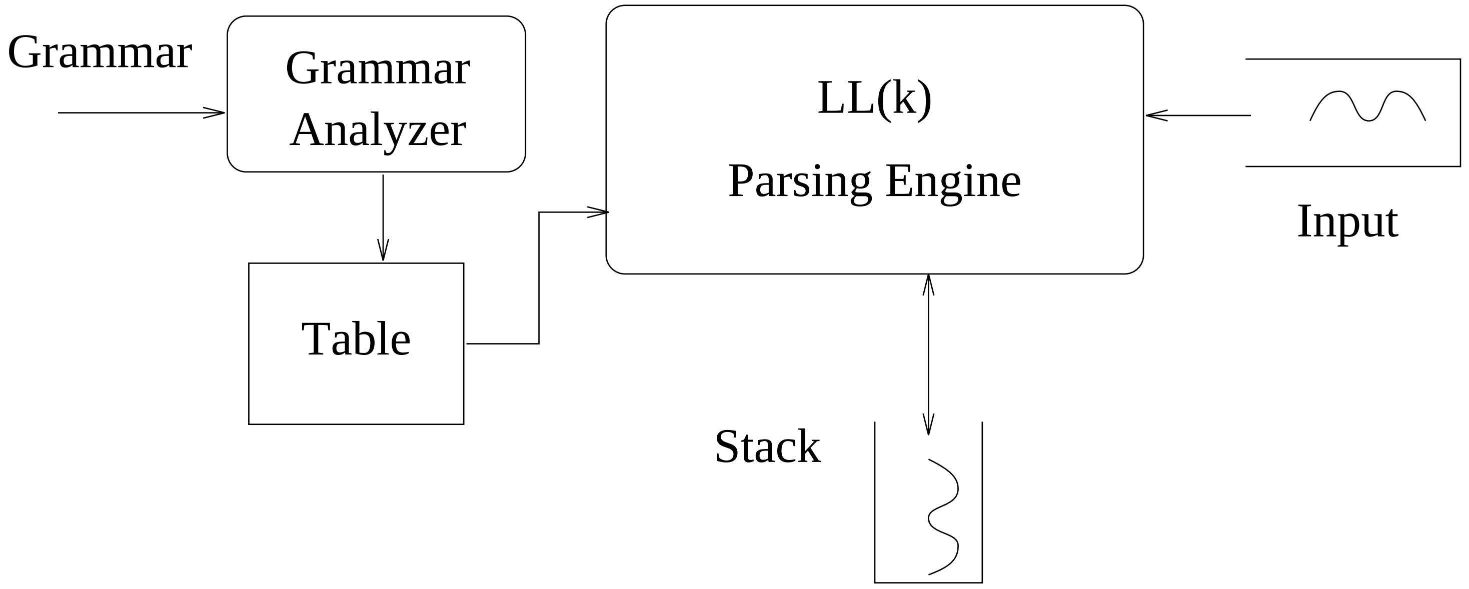
	First	**Follow**
E	$\{ (, a \}$	$\{), \$ \}$
E'	$\{ +, - \}$	$\{), \$ \}$
T	$\{ (, a \}$	$\{ +, -,), \$ \}$
T'	$\{ *, / \}$	$\{ +, -,), \$ \}$
F	$\{ (, a \}$	$\{ *, /, +, -,), \$ \}$

```
Procedure $E'(StopSet)$
    if $(LookAhead(+))$ then
        call $Expect(+)$
        call $T(\{+,-\} \cup StopSet)$
        call $E'(StopSet)$
    else
        if $(LookAhead(\$, ')'))$ then
            return ()
        else
            call $ErrorRecover(StopSet)$
        fi
    fi
end
```

Table-driven $LL(k)$ parsing

Our recursive descent parser contained a procedure for each nonterminal. The generation of these procedures could be automated—through the construction and testing of $First$ and $Follow$ sets—for any grammar free of left recursion.

Another equally automatable approach is to use a simple *parsing engine* **that is driven by tables constructed by similar analysis of the grammar.**



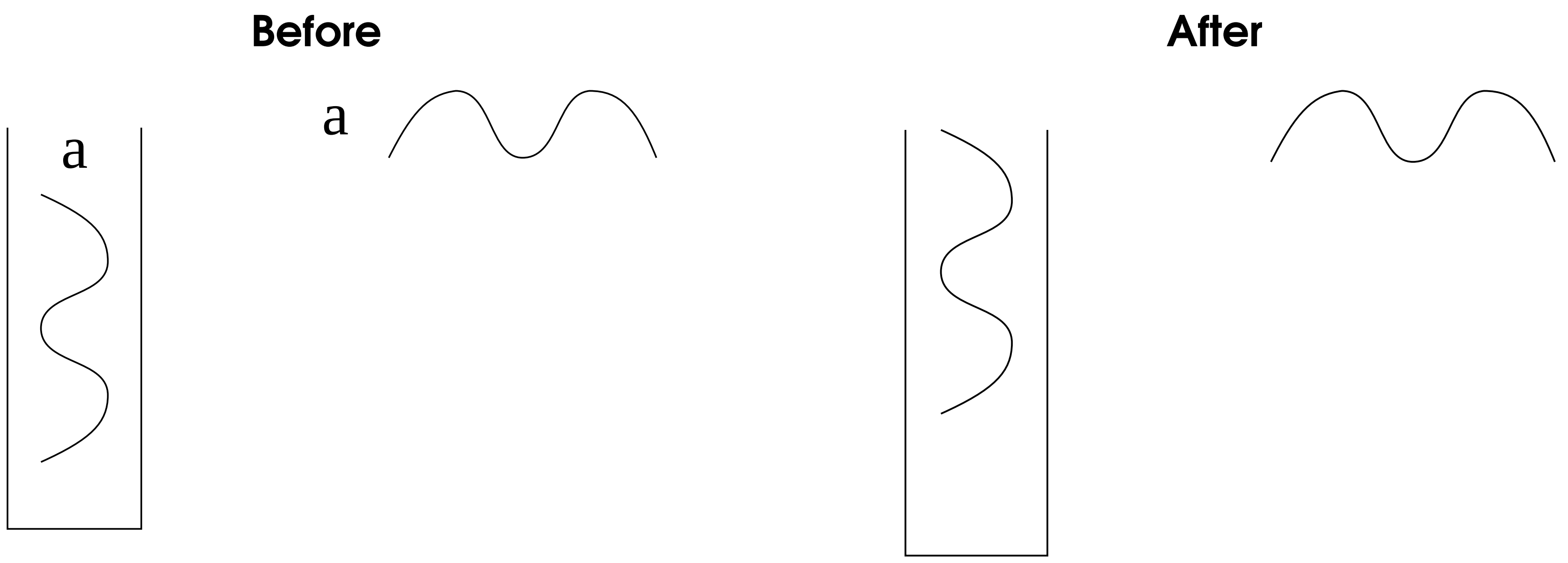
The parsing engine begins by pushing the start symbol S onto the stack. Each subsequent action is one of the following:

Match: pairs an input symbol a an a on top-of-stack.

Apply: replaces the nonterminal N with ω, where $(N \rightarrow \omega) \in P$.

Match

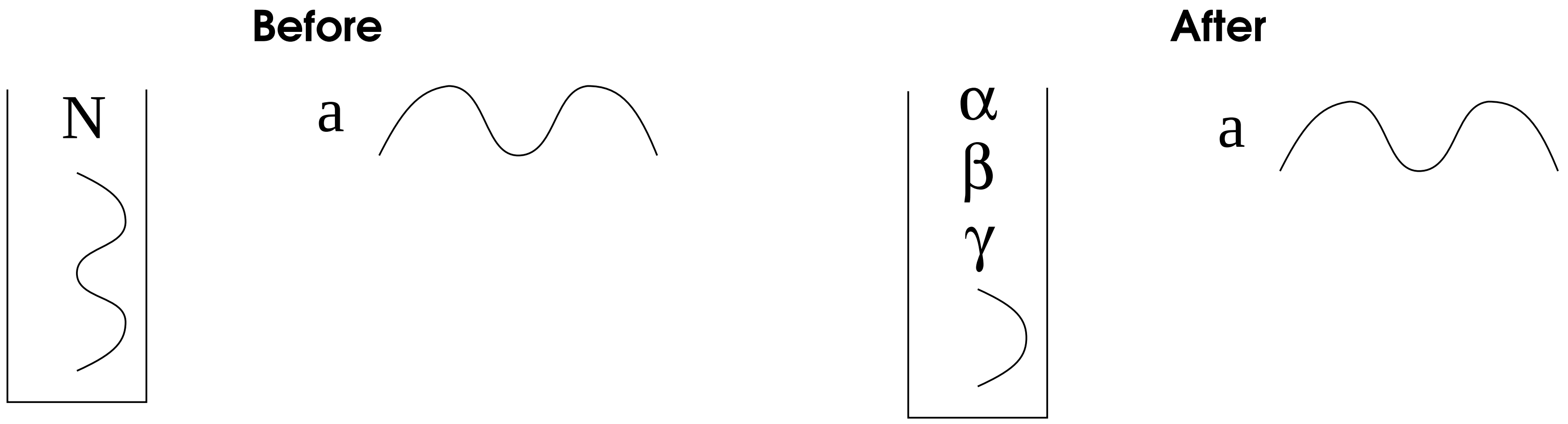
If the top-of-stack contains the terminal symbol "a", then the parsing engine must find an "a" as the next input symbol; the stack is popped, and the input is advanced.



- If a match simultaneously empties the stack and exhausts the input stream, then the string is *accepted* by the parser.
- If a match is attempted, but the symbols disagree, then an error is declared.

Apply

If the top-of-stack contains a nonterminal N, then the parsing engine must choose the appropriate rule for N, say $N \to \alpha\beta\gamma$. The stack is popped of symbol N, and the symbols α, β, and γ are pushed onto the stack, such that α is the new top-of-stack.



Since a match is always required when a terminal is exposed on top-of-stack, the only information that must be coded in our table is the rule that should be applied when a nonterminal appears on top-of-stack. As with our recursive descent parser, this decision can be based on k symbols of lookahead into the input stream.

Constructing the table

$\boxed{1}$	**S**	$\rightarrow$	**A C $**
$\boxed{2}$	**C**	$\rightarrow$	**c**
$\boxed{3}$		$\mid$	λ
$\boxed{4}$	**A**	$\rightarrow$	**a B C d**
$\boxed{5}$		$\mid$	**B Q**
$\boxed{6}$		$\mid$	λ
$\boxed{7}$	**B**	$\rightarrow$	**b B**
$\boxed{8}$		$\mid$	**d**
$\boxed{9}$	**Q**	$\rightarrow$	**q**

	First	**Follow**
S	$\{a, b, d, c, \$\}$	$\{\ \}$
A	$\{a, b, d, \lambda\}$	$\{c, \$\}$
B	$\{b, d\}$	$\{c, d, q\}$
C	$\{c, \lambda\}$	$\{d, \$\}$
Q	$\{q\}$	$\{c, \$\}$

	Lookahead					
NonTerm	a	b	c	d	q	$
S	1	1	1	1		1
C			2	3		3
A	4	5	6	5		6
B		7		8		
Q					9	

The nonblank entries in the above table indicate the number of the rule that should be applied, given a nonterminal on top-of-stack and an input symbol as lookahead.

Using the table

$\boxed{1}$	**S**	$\rightarrow$	**A C $**
$\boxed{2}$	**C**	$\rightarrow$	**c**
$\boxed{3}$		$\vert$	λ
$\boxed{4}$	**A**	$\rightarrow$	**a B C d**
$\boxed{5}$		$\vert$	**B Q**
$\boxed{6}$		$\vert$	λ
$\boxed{7}$	**B**	$\rightarrow$	**b B**
$\boxed{8}$		$\vert$	**d**
$\boxed{9}$	**Q**	$\rightarrow$	**q**

	Lookahead					
NonTerm	**a**	**b**	**c**	**d**	**q**	**$**
S	1	1	1	1		1
C			2	3		3
A	4	5	6	5		6
B		7		8		
Q					9	

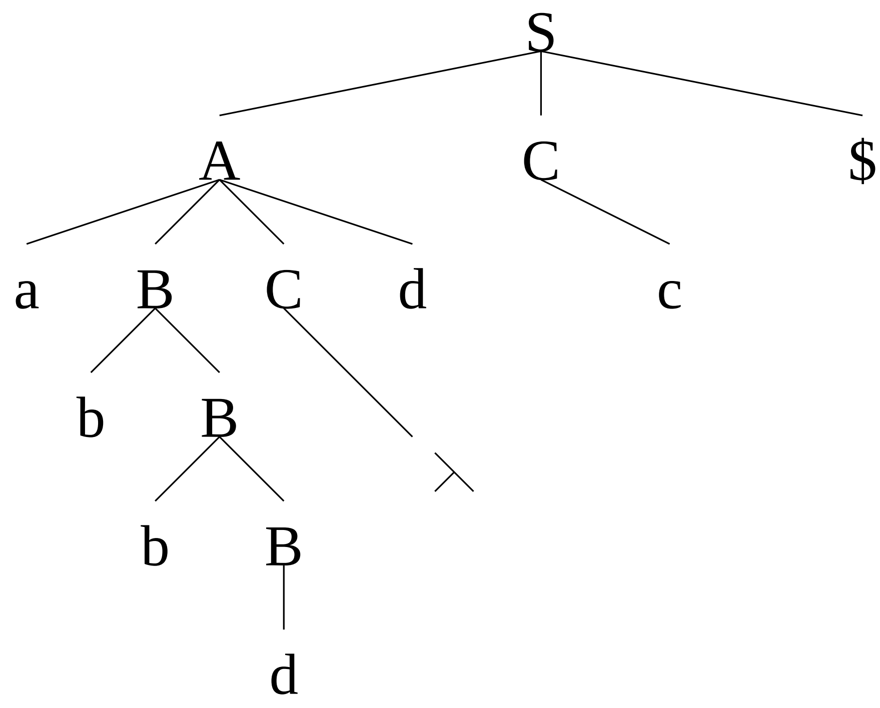
Below is shown the stack activity in parsing the input string "abbddc$".

		a	b	b			
		B	B	B	d		
		C	C	C	C		
	A	d	d	d	d	d	
	C	C	C	C	C	C	c
S	$	$	$	$	$	$	$
	$\boxed{1}$	$\boxed{4}$	$\boxed{7}$	$\boxed{7}$	$\boxed{8}$	$\boxed{3}$	$\boxed{2}$

Input string

a b b d d c $

Bottom-up parsing



$\boxed{1}$	**S**	$\rightarrow$	**A C $**
$\boxed{2}$	**C**	$\rightarrow$	**c**
$\boxed{3}$		$\mid$	λ
$\boxed{4}$	**A**	$\rightarrow$	**a B C d**
$\boxed{5}$		$\mid$	**B Q**
$\boxed{6}$		$\mid$	λ
$\boxed{7}$	**B**	$\rightarrow$	**b B**
$\boxed{8}$		$\mid$	**d**
$\boxed{9}$	**Q**	$\rightarrow$	**q**

$$\begin{aligned}
\mathbf{S} &\Rightarrow \mathbf{A}\ \mathit{C}\ \mathbf{\$} \\
&\Rightarrow \mathit{A}\ \boxed{\mathbf{c}}\ \mathbf{\$} \\
&\Rightarrow \boxed{\mathbf{a}\ \mathbf{B}\ \mathit{C}\ \mathbf{d}}\ \mathbf{c}\ \mathbf{\$} \\
&\Rightarrow \mathbf{a}\ \mathit{B}\ \boxed{}\ \mathbf{d}\ \mathbf{c}\ \mathbf{\$} \\
&\Rightarrow \mathbf{a}\ \boxed{\mathbf{b}\ \mathit{B}}\ \mathbf{d}\ \mathbf{c}\ \mathbf{\$} \\
&\Rightarrow \mathbf{a}\ \mathbf{b}\ \boxed{\mathbf{b}\ \mathit{B}}\ \mathbf{d}\ \mathbf{c}\ \mathbf{\$} \\
&\Rightarrow \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \boxed{\mathbf{d}}\ \mathbf{d}\ \mathbf{c}\ \mathbf{\$}
\end{aligned}$$

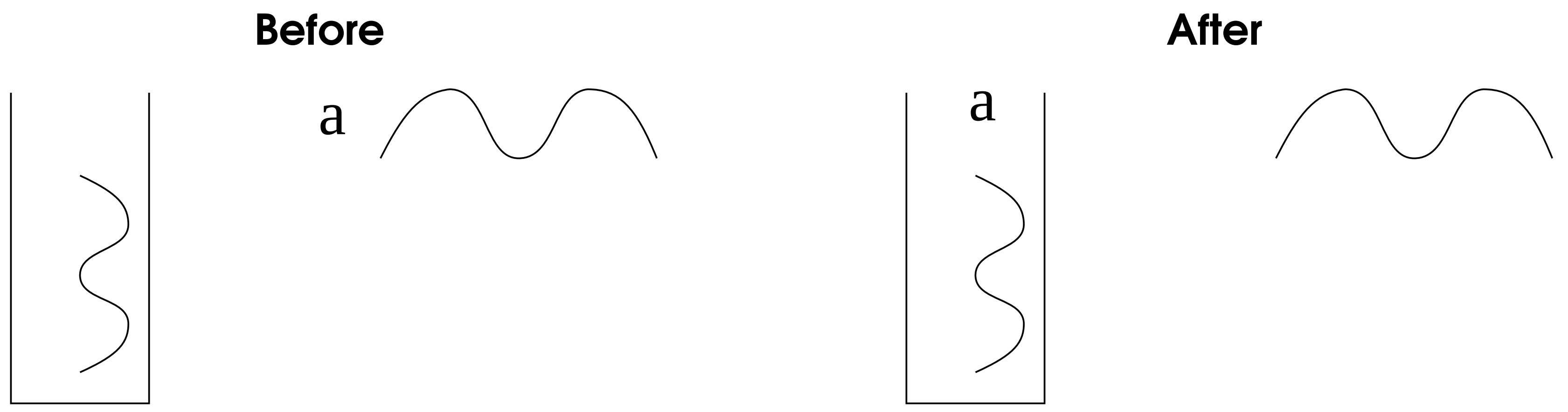
A bottom-up parse is essentially a right-most derivation, run in *reverse*. Instead of replacing a nonterminal by a string, we recognize the string as *reducing* to the nonterminal.

The parsing engine issues the following instructions:

shift: a symbol is moved from input to top-of-stack.

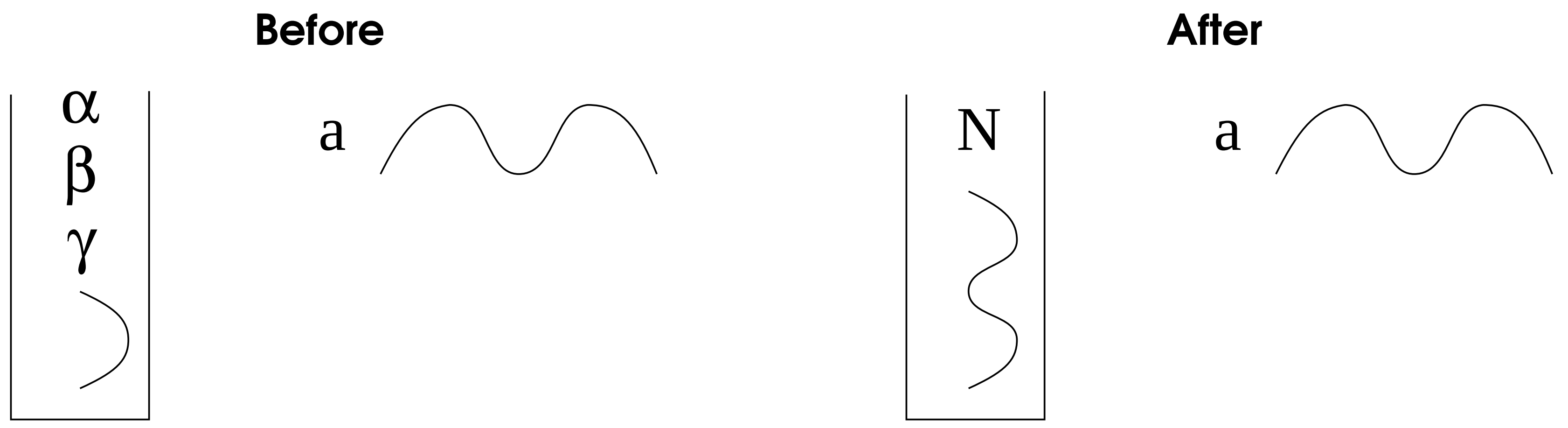
reduce r**:** the stack is modified by applying rule r.

Shift



- Like the top-down parser, the bottom-up parser checks for errors on a shift. The parse table we shall construct indicates when a shift is error-free.
- Actually, instead of pushing a symbol onto the stack, we push a *state*, which indexes the parse table and represents the current possibilities of the parse.

Reduce



- If the rule applied is $N \rightarrow \omega$, where ω has m symbols, then m symbols are popped off the stack, and a symbol representing N is pushed.
- It's important to remember that a canonical parse can perform reductions *only* at the top-of-stack.

Rightmost derivation in reverse – slow motion

Stack	**Input**	**Activity**
	a b b d d c $	Shift
a	b b d d c $	Shift
a b	b d d c $	Shift
a b b	d d c $	Shift
a b b d	d c $	Reduce $B \rightarrow d$
a b b B	d c $	Reduce $B \rightarrow bB$
a b B	d c $	Reduce $B \rightarrow bB$
a B	d c $	Reduce $C \rightarrow \lambda$
a B C	d c $	Shift
a B C d	c $	Reduce $A \rightarrow aBCd$
A	c $	Shift
A c	$	Reduce $C \rightarrow c$
A C	$	Shift
A C $		Reduce $S \rightarrow AC\$$
S		Accept

This is LR-style parsing: a scan from the left that produces a rightmost derivation.

We could have tried to apply $C \rightarrow \lambda$ at any point during the parse, but most would not have made progress toward an accept. Where parse table construction is successful, the table directs the parse towards an accept if one is possible.

LR table construction

Each state of the parser represents parsing possibilities after processing a given prefix of the input string.

To construct the canonical $LR(0)$ set of states:

1. **Each state begins with a *kernel* that represents progress through certain rules of the grammar:**

(3)	X	$\rightarrow$	y	•	z
	W	$\rightarrow$	x z y	•	A
	F	$\rightarrow$	a B C y	•	

   The dot (•) shows the progress through the rule achieved by moving into this state.

2. **When • is next to a nonterminal, we must add into this state the *closure* by expanding all rules of the nonterminal:**

(3)	A	$\rightarrow$	•	b c d
	A	$\rightarrow$	•	z A

We then label each component of the state with an action, indicating transfer to some other state, reduction by a rule, or accept:

(3)	X	$\rightarrow$	y	•	z	Goto State	17
	W	$\rightarrow$	x z y	•	A	Goto State	5
	F	$\rightarrow$	a B C y	•		Reduce by rule	$\boxed{5}$
	A	$\rightarrow$		•	b c d	Goto State	2
	A	$\rightarrow$		•	z A	Goto State	17

which may create a new state:

(17)	X	$\rightarrow$	y z	•		Reduce by rule	$\boxed{10}$
	A	$\rightarrow$	z	•	A	Goto State	1
	A	$\rightarrow$		•	b c d	Goto State	2
	A	$\rightarrow$		•	z A	Goto State	18

Table construction

(1)	S	→	• A C $	Goto State	2
	A	→	• a B C d	Goto State	3
		|	• B Q	Goto State	4
		|	•	Reduce by rule	$\boxed{6}$
	B	→	• b B	Goto State	5
		|	• d	Goto State	6

(2)	S	→	A • C $	Goto State	7
	C	→	• c	Goto State	8
		|	•	Reduce by rule	$\boxed{3}$

(3)	A	→	a • B C d	Goto State	9
	B	→	• b B	Goto State	5
		|	• d	Goto State	6

(4)	A	→	B • Q	Goto State	10
	Q	→	• q	Goto State	11

(5)	B	→	b • B	Goto State	12
	B	→	• b B	Goto State	5
		|	• d	Goto State	6

(6)	B	→	d •	Reduce by rule	$\boxed{8}$

(7)	S	→	A C • $	Goto State	13

(8)	C	→	c •	Reduce by rule	$\boxed{2}$

(9)	A	→	a B • C d	Goto State	14
	C	→	• c	Goto State	8
		|	•	Reduce by rule	$\boxed{3}$

(10)	A	→	B Q •	Reduce by rule	$\boxed{5}$

(11)	Q	→	q •	Reduce by rule	$\boxed{9}$

(12)	B	→	b B •	Reduce by rule	$\boxed{7}$

(13)	S	→	A C $ •		☺

(14)	A	→	a B C • d	Goto State	15

(15)	A	→	a B C d •	Reduce by rule	$\boxed{4}$

Conflict resolution

$\boxed{1}$	S	$\rightarrow$	A C \$
$\boxed{2}$	C	$\rightarrow$	c
$\boxed{3}$		$\mid$	λ
$\boxed{4}$	A	$\rightarrow$	a B C d
$\boxed{5}$		$\mid$	B Q
$\boxed{6}$		$\mid$	λ
$\boxed{7}$	B	$\rightarrow$	b B
$\boxed{8}$		$\mid$	d
$\boxed{9}$	Q	$\rightarrow$	q

	First	Follow
S	$\{a, b, d, c, \$\}$	$\{\}$
A	$\{a, b, d, \lambda\}$	$\{c, \$\}$
B	$\{b, d\}$	$\{c, d, q\}$
C	$\{c, \lambda\}$	$\{d, \$\}$
Q	$\{q\}$	$\{c, \$\}$

Within a state, how do we resolve whether to shift or reduce when either action seems appropriate?

(1)	**S**	$\rightarrow$	$\bullet$ **A C \$**	Goto State **2**
	A	$\rightarrow$	$\bullet$ **a B C d**	Goto State **3**
		$\mid$	$\bullet$ **B Q**	Goto State **4**
		$\mid$	$\bullet$	Reduce by rule $\boxed{6}$
	B	$\rightarrow$	$\bullet$ **b B**	Goto State **5**
		$\mid$	$\bullet$ **d**	Goto State **6**

Examining the $Follow$ information shows that only those input symbols in $\{c, \$\}$ can follow an A. In state (1) we therefore Reduce by rule $\boxed{6}$ **only when "c" or "\$" appears next in the input. Since these symbols are disjoint from the input symbols that cause shifts into other states ($\{a, b, d\}$), we can resolve the apparent conflict.**

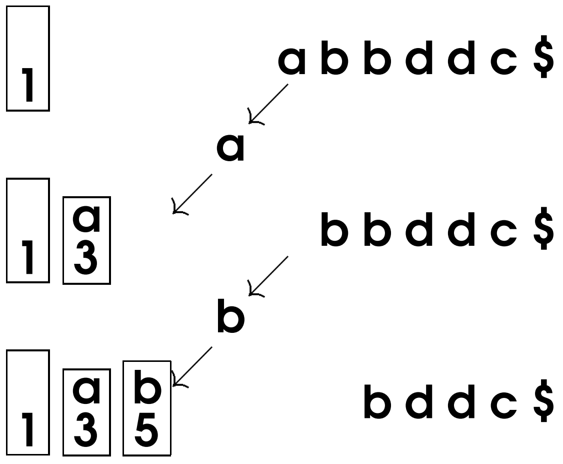
In general, a state might have an apparent shift/reduce or reduce/reduce conflict. The more expensive table construction methods generally provide better conflict resolution.

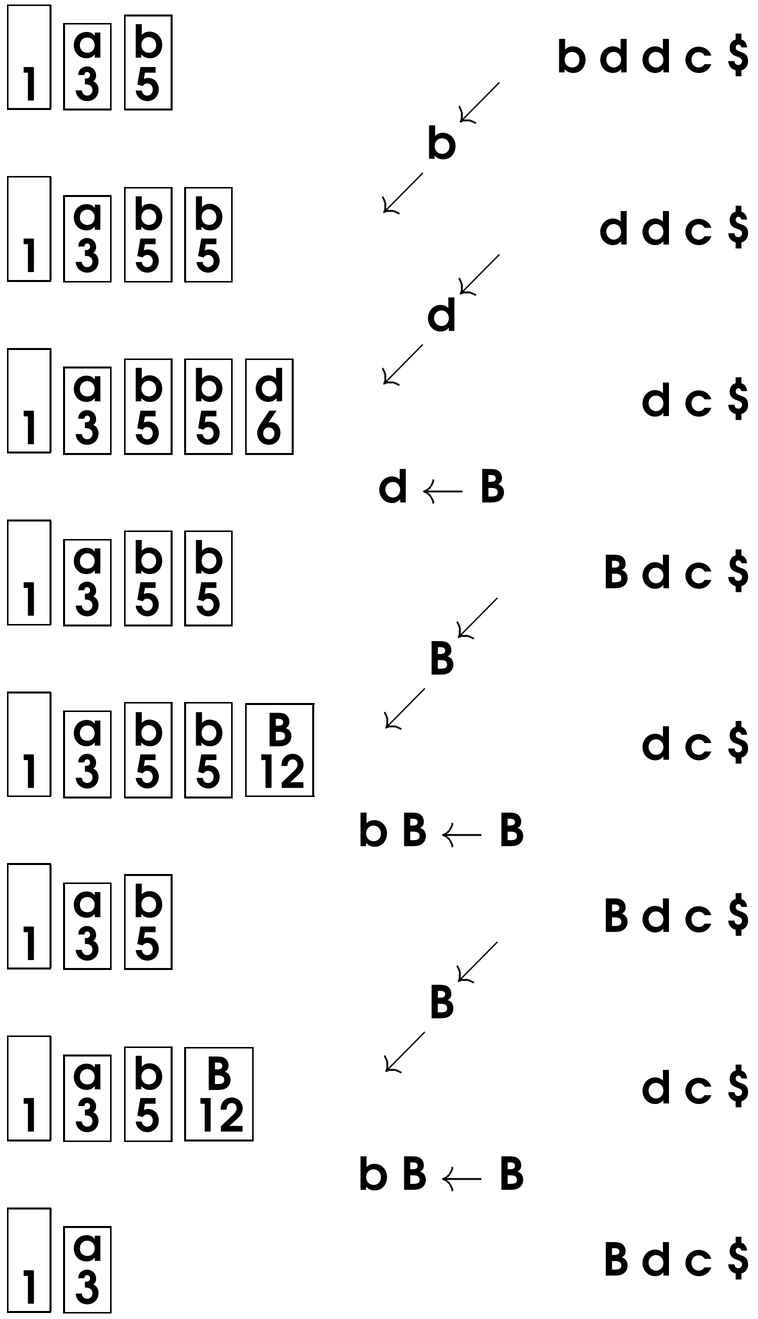
Table for our example

State	a	b	c	d	q	\$	A	B	C	Q
1	Goto State **3**	Goto State **5**	Reduce by rule $\boxed{6}$	Goto State **6**		Reduce by rule $\boxed{6}$	Goto State **2**	Goto State **4**		
2			Goto State **8**	Reduce by rule $\boxed{3}$		Reduce by rule $\boxed{3}$			Goto State **7**	
3		Goto State **5**		Goto State **6**				Goto State **9**		
4					Goto State **11**					Goto State **10**
5		Goto State **5**		Goto State **6**				Goto State **12**		
6	Reduce by rule $\boxed{8}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	Reduce by rule $\boxed{8}$
7						Goto State **13**				
8	Reduce by rule $\boxed{2}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	Reduce by rule $\boxed{2}$
9			Goto State **8**	Reduce by rule $\boxed{3}$		Reduce by rule $\boxed{3}$			Goto State **14**	
10	Reduce by rule $\boxed{5}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	Reduce by rule $\boxed{5}$
11	Reduce by rule $\boxed{9}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	Reduce by rule $\boxed{9}$
12	Reduce by rule $\boxed{7}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	Reduce by rule $\boxed{7}$
13	☺	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	☺
14				Goto State **15**						
15	Reduce by rule $\boxed{4}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	Reduce by rule $\boxed{4}$

Using the table

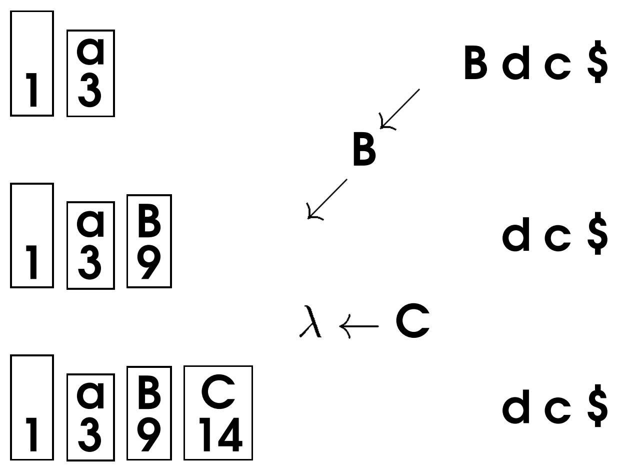
State	a	b	c	d	q	$	A	B	C	Q
1	3	5	$\boxed{6}$	6		$\boxed{6}$	2	4		
2			8	$\boxed{3}$		$\boxed{3}$			7	
3		5		6				9		
4					11					10
5		5		6				12		
6	$\boxed{8}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	$\boxed{8}$
7						13				
8	$\boxed{2}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	$\boxed{2}$
9			8	$\boxed{3}$		$\boxed{3}$			14	
10	$\boxed{5}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	$\boxed{5}$
11	$\boxed{9}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	$\boxed{9}$
12	$\boxed{7}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	$\boxed{7}$
13	☺	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	☺
14				15						
15	$\boxed{4}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	$\boxed{4}$

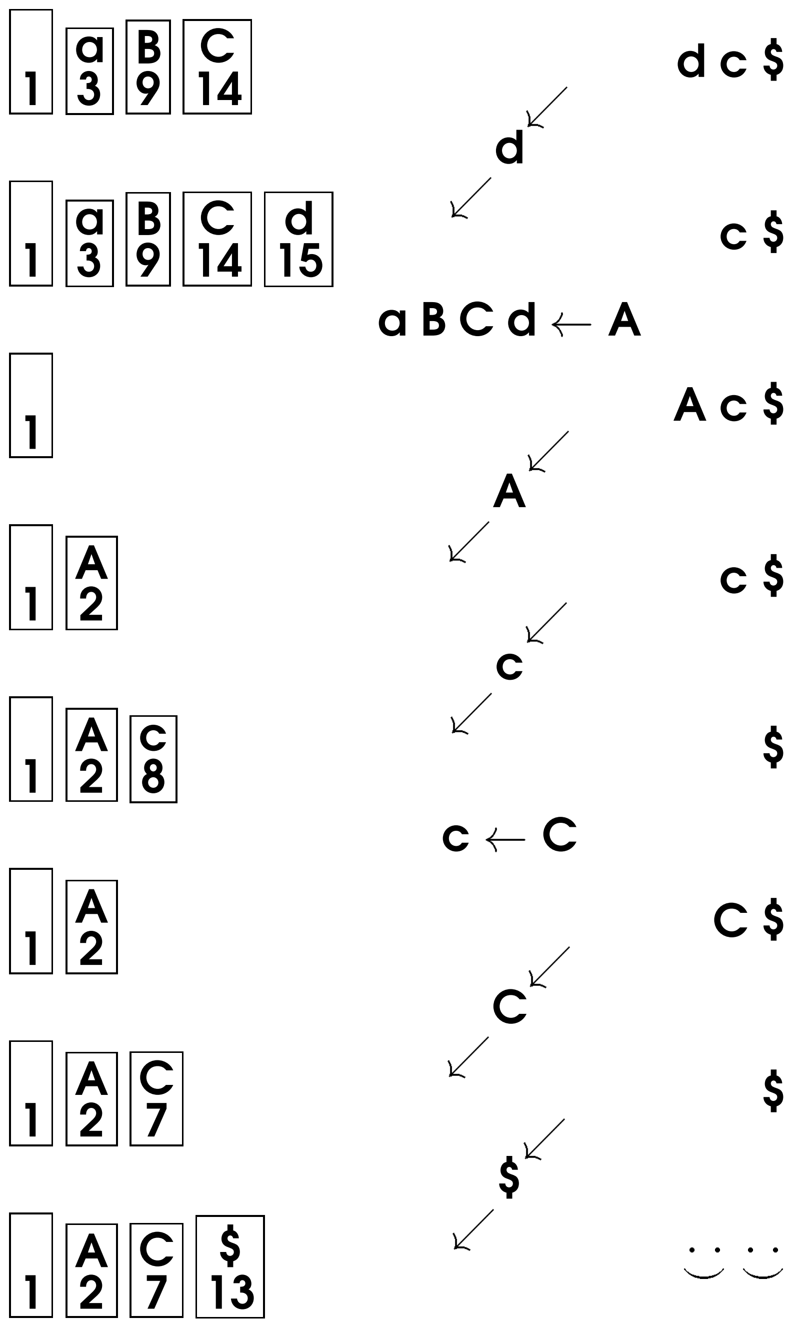




Using the table (cont'd)

State	a	b	c	d	q	$	A	B	C	Q
1	3	5	$\boxed{6}$	6		$\boxed{6}$	2	4		
2			8	$\boxed{3}$		$\boxed{3}$			7	
3		5		6				9		
4					11					10
5		5		6				12		
6	$\boxed{8}$	…	…	…	…	…	…	…	…	$\boxed{8}$
7						13				
8	$\boxed{2}$	…	…	…	…	…	…	…	…	$\boxed{2}$
9			8	$\boxed{3}$		$\boxed{3}$			14	
10	$\boxed{5}$	…	…	…	…	…	…	…	…	$\boxed{5}$
11	$\boxed{9}$	…	…	…	…	…	…	…	…	$\boxed{9}$
12	$\boxed{7}$	…	…	…	…	…	…	…	…	$\boxed{7}$
13	☺	…	…	…	…	…	…	…	…	☺
14				15						
15	$\boxed{4}$	…	…	…	…	…	…	…	…	$\boxed{4}$





Set of items construction for our expression grammar

$\boxed{1}$	S	$\rightarrow$	E \$
$\boxed{2}$	E	$\rightarrow$	E + T
$\boxed{3}$		$\mid$	T
$\boxed{4}$	T	$\rightarrow$	T $*$ F
$\boxed{5}$		$\mid$	F
$\boxed{6}$	F	$\rightarrow$	(E)
$\boxed{7}$		$\mid$	a

	First	Follow
S	$\{(, a\}$	$\{\}$
E	$\{(, a\}$	$\{+,), \$\}$
T	$\{(, a\}$	$\{*, +,), \$\}$
F	$\{(, a\}$	$\{*, +,), \$\}$

(1)	S	$\rightarrow$	$\bullet$ E \$	Goto State **2**
	E	$\rightarrow$	$\bullet$ E + T	Goto State **2**
		$\mid$	$\bullet$ T	Goto State **3**
	T	$\rightarrow$	$\bullet$ T $*$ F	Goto State **3**
		$\mid$	$\bullet$ F	Goto State **4**
	F	$\rightarrow$	$\bullet$ (E)	Goto State **5**
		$\mid$	$\bullet$ a	Goto State **6**

(2)	S	$\rightarrow$	E $\bullet$ \$	Goto State **7**
	E	$\rightarrow$	E $\bullet$ + T	Goto State **8**

(3)	E	$\rightarrow$	T $\bullet$	Reduce by rule $\boxed{3}$
	T	$\rightarrow$	T $\bullet$ $*$ F	Goto State **9**

The above shift/reduce conflict is resolved by noting that $* \notin Follow(E)$.

(4)	T	$\rightarrow$	F $\bullet$	Reduce by rule $\boxed{5}$

(5)	F	$\rightarrow$	($\bullet$ E)	Goto State **10**
	E	$\rightarrow$	$\bullet$ E + T	Goto State **10**
		$\mid$	$\bullet$ T	Goto State **3**
	T	$\rightarrow$	$\bullet$ T $*$ F	Goto State **3**
		$\mid$	$\bullet$ F	Goto State **4**
	F	$\rightarrow$	$\bullet$ (E)	Goto State **5**
		$\mid$	$\bullet$ a	Goto State **6**

(6)	F	$\rightarrow$	a $\bullet$	Reduce by rule $\boxed{7}$

Set of items construction for our expression grammar

$\boxed{1}$	**S**	$\rightarrow$	**E $**
$\boxed{2}$	**E**	$\rightarrow$	**E + T**
$\boxed{3}$		$\vert$	**T**
$\boxed{4}$	**T**	$\rightarrow$	**T $*$ F**
$\boxed{5}$		$\vert$	**F**
$\boxed{6}$	**F**	$\rightarrow$	**(E)**
$\boxed{7}$		$\vert$	**a**

	First	**Follow**
S	$\{\,(, a\,\}$	$\{\ \}$
E	$\{\,(, a\,\}$	$\{\,+,), \$\,\}$
T	$\{\,(, a\,\}$	$\{\,*, +,), \$\,\}$
F	$\{\,(, a\,\}$	$\{\,*, +,), \$\,\}$

(7)	**S**	$\rightarrow$	**E $** $\bullet$	☺

(8)	**E**	$\rightarrow$	**E +** $\bullet$ **T**	Goto State **11**
	T	$\rightarrow$	$\bullet$ **T $*$ F**	Goto State **11**
		$\vert$	$\bullet$ **F**	Goto State **4**
	F	$\rightarrow$	$\bullet$ **(E)**	Goto State **5**
		$\vert$	$\bullet$ **a**	Goto State **6**

(9)	**T**	$\rightarrow$	**T $*$** $\bullet$ **F**	Goto State **12**
	F	$\rightarrow$	$\bullet$ **(E)**	Goto State **5**
		$\vert$	$\bullet$ **a**	Goto State **6**

(10)	**E**	$\rightarrow$	**E** $\bullet$ **+ T**	Goto State **8**
	F	$\rightarrow$	**(E** $\bullet$ **)**	Goto State **13**

(11)	**E**	$\rightarrow$	**E + T** $\bullet$	Reduce by rule $\boxed{2}$
	T	$\rightarrow$	**T** $\bullet$ **$*$ F**	Goto State **9**

The above shift/reduce conflict is resolved by noting that $* \notin Follow(E)$.

(12)	**T**	$\rightarrow$	**T $*$ F** $\bullet$	Reduce by rule $\boxed{4}$

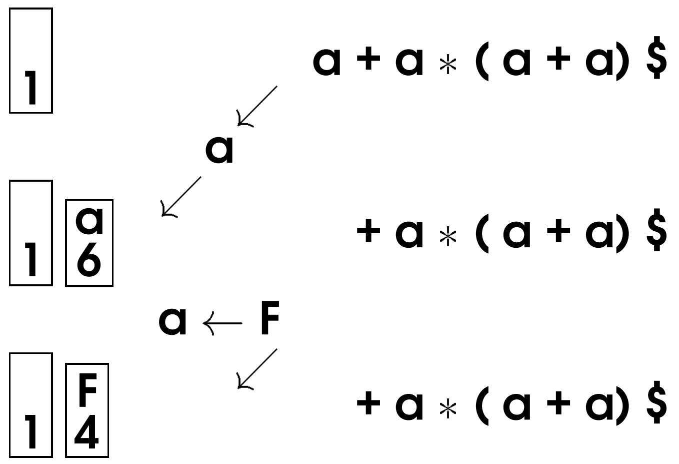
(13)	**F**	$\rightarrow$	**(E)** $\bullet$	Reduce by rule $\boxed{6}$

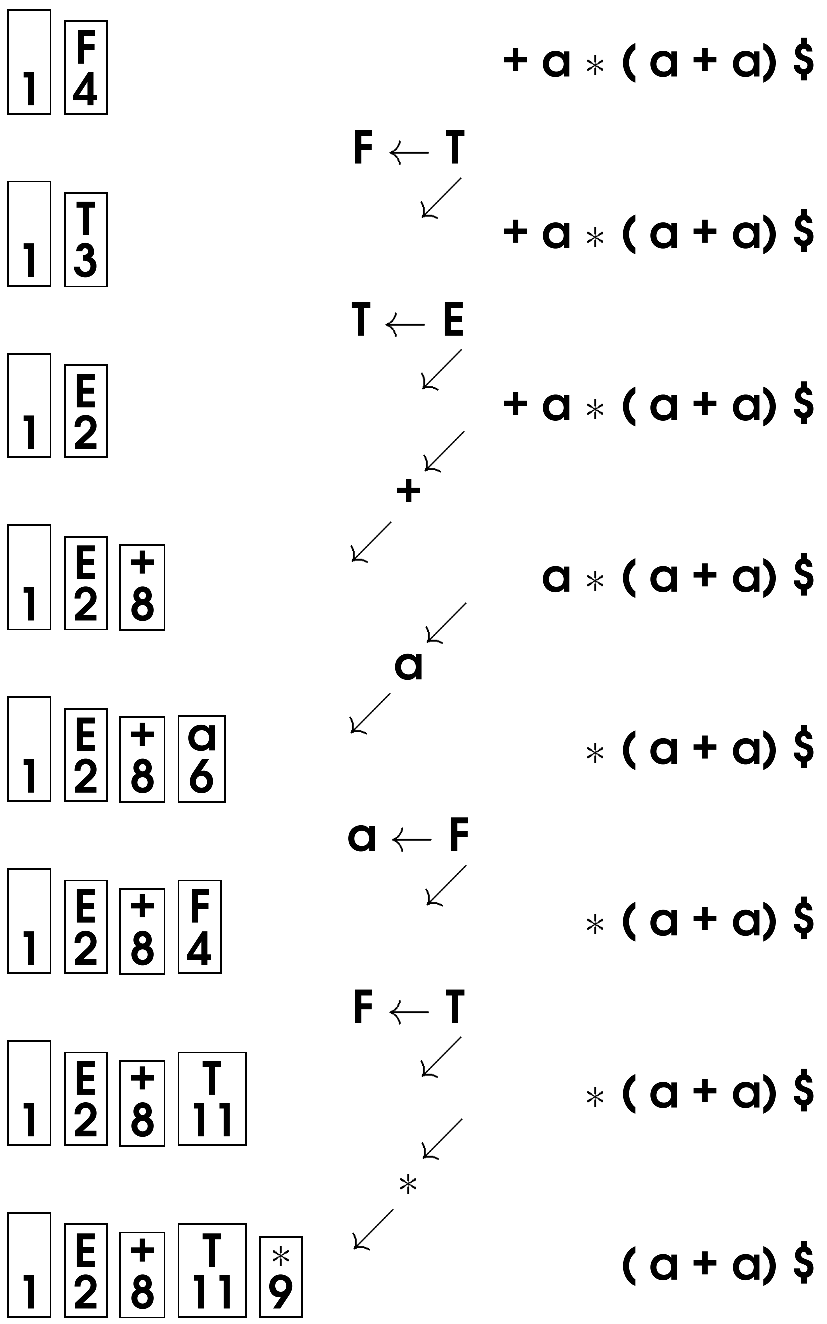
The resulting parse table

State	a	+	*	(	)	$	E	T	F
1	Goto State **6**			Goto State **5**			Goto State **2**	Goto State **3**	Goto State **4**
2		Goto State **8**				Goto State **7**			
3		Reduce by rule $\boxed{3}$	Goto State **9**		Reduce by rule $\boxed{3}$	Reduce by rule $\boxed{3}$			
4	Reduce by rule $\boxed{5}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	Reduce by rule $\boxed{5}$
5	Goto State **6**			Goto State **5**			Goto State **10**	Goto State **3**	Goto State **4**
6	Reduce by rule $\boxed{7}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	Reduce by rule $\boxed{7}$
7	☺	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	☺
8	Goto State **6**			Goto State **5**				Goto State **11**	Goto State **4**
9	Goto State **6**			Goto State **5**					Goto State **12**
10		Goto State **8**			Goto State **13**				
11		Reduce by rule $\boxed{2}$	Goto State **9**		Reduce by rule $\boxed{2}$	Reduce by rule $\boxed{2}$			
12	Reduce by rule $\boxed{4}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	Reduce by rule $\boxed{4}$
13	Reduce by rule $\boxed{6}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	Reduce by rule $\boxed{6}$

Using the table

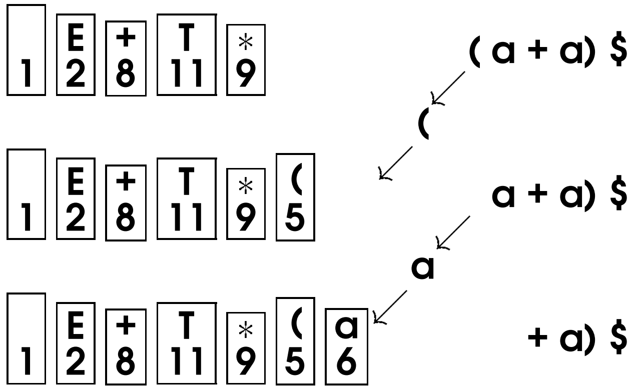
State	a	+	*	(	)	$	E	T	F
1	6			5			2	3	4
2		8				7			
3		$\boxed{3}$	9		$\boxed{3}$	$\boxed{3}$			
4	$\boxed{5}$	· ·	· ·	· ·	· ·	· ·	· ·	· ·	$\boxed{5}$
5	6			5			10	3	4
6	$\boxed{7}$	· ·	· ·	· ·	· ·	· ·	· ·	· ·	$\boxed{7}$
7	☺	· ·	· ·	· ·	· ·	· ·	· ·	· ·	☺
8	6			5				11	4
9	6			5					12
10		8			13				
11		$\boxed{2}$	9		$\boxed{2}$	$\boxed{2}$			
12	$\boxed{4}$	· ·	· ·	· ·	· ·	· ·	· ·	· ·	$\boxed{4}$
13	$\boxed{6}$	· ·	· ·	· ·	· ·	· ·	· ·	· ·	$\boxed{6}$

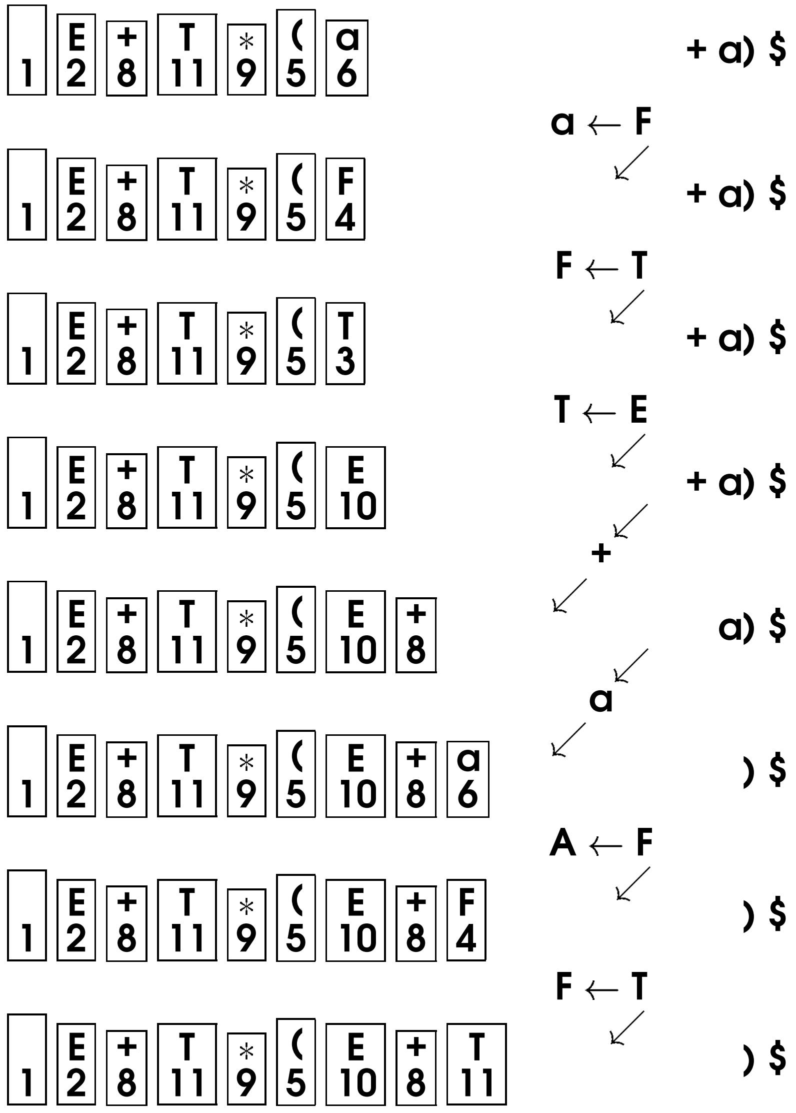




Using the table

State	a	+	*	(	)	$	E	T	F
1	6			5			2	3	4
2		8				7			
3		$\boxed{3}$	9		$\boxed{3}$	$\boxed{3}$			
4	$\boxed{5}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	$\boxed{5}$
5	6			5			10	3	4
6	$\boxed{7}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	$\boxed{7}$
7	☺	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	☺
8	6			5				11	4
9	6			5					12
10		8			13				
11		$\boxed{2}$	9		$\boxed{2}$	$\boxed{2}$			
12	$\boxed{4}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	$\boxed{4}$
13	$\boxed{6}$	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	$\boxed{6}$





Using the table

State	a	+	$*$	(	)	$	E	T	F
1	6			5			2	3	4
2		8				7			
3		$\boxed{3}$	9		$\boxed{3}$	$\boxed{3}$			
4	$\boxed{5}$	· ·	· ·	· ·	· ·	· ·	· ·	· ·	$\boxed{5}$
5	6			5			10	3	4
6	$\boxed{7}$	· ·	· ·	· ·	· ·	· ·	· ·	· ·	$\boxed{7}$
7	☺	· ·	· ·	· ·	· ·	· ·	· ·	· ·	☺
8	6			5				11	4
9	6			5					12
10		8			13				
11		$\boxed{2}$	9		$\boxed{2}$	$\boxed{2}$			
12	$\boxed{4}$	· ·	· ·	· ·	· ·	· ·	· ·	· ·	$\boxed{4}$
13	$\boxed{6}$	· ·	· ·	· ·	· ·	· ·	· ·	· ·	$\boxed{6}$

$\boxed{1}\boxed{\substack{E\\2}}\boxed{\substack{+\\8}}\boxed{\substack{T\\11}}\boxed{\substack{*\\9}}\boxed{\substack{(\\5}}\boxed{\substack{E\\10}}\boxed{\substack{+\\8}}\boxed{\substack{T\\11}}$) $

E + T ← E ↙

$\boxed{1}\boxed{\substack{E\\2}}\boxed{\substack{+\\8}}\boxed{\substack{T\\11}}\boxed{\substack{*\\9}}\boxed{\substack{(\\5}}\boxed{\substack{E\\10}}$) $

↙) ↙

$\boxed{1}\boxed{\substack{E\\2}}\boxed{\substack{+\\8}}\boxed{\substack{T\\11}}\boxed{\substack{*\\9}}\boxed{\substack{(\\5}}\boxed{\substack{E\\10}}\boxed{\substack{)\\13}}$ $

$\boxed{1}\boxed{\substack{E\\2}}\boxed{\substack{+\\8}}\boxed{\substack{T\\11}}\boxed{\substack{*\\9}}\boxed{\substack{(\\5}}\boxed{\substack{E\\10}}\boxed{\substack{)\\13}}$ $

(E) ← F ↙

$\boxed{1}\boxed{\substack{E\\2}}\boxed{\substack{+\\8}}\boxed{\substack{T\\11}}\boxed{\substack{*\\9}}\boxed{\substack{F\\12}}$ $

T $*$ F ← T ↙

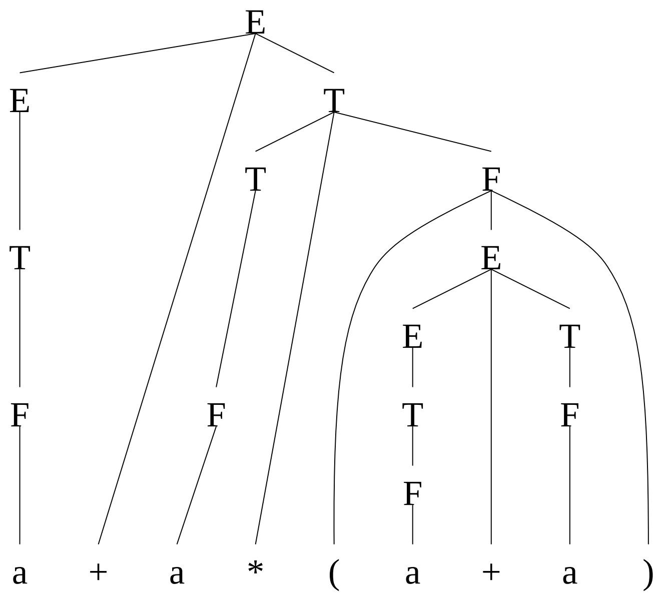
$\boxed{1}\boxed{\substack{E\\2}}\boxed{\substack{+\\8}}\boxed{\substack{T\\11}}$ $

E + T ← E ↙

$\boxed{1}\boxed{\substack{E\\2}}$ $

↙ $ ↙

$\boxed{1}\boxed{\substack{E\\2}}\boxed{\substack{\$\\7}}$ ☺ ☺



Summary of LR table construction methods

LR(0): If the table contains no conflicts, then the grammar is unambiguous and each state clearly indicates precise shifts and reduces.

SLR(k): Where conflicts exist, this method analyzes the grammar to obtain sets of at the k symbols that can follow each nonterminal. For an item containing

$$\boxed{\begin{array}{llll} (1) & A & \rightarrow & B\,C\,\bullet \\ & D & \rightarrow & C\,\bullet\,F \\ & G & \rightarrow & C\,\bullet \end{array}}$$

if the k symbols that can follow A are disjoint from each of the strings of k symbols derivable from F, then the shift/reduce conflict is resolved. If the k symbols that can follow A are different from those that can follow G, then the reduce/reduce conflict is resolved.

LR(k): While the SLR method analyzes the grammar for follow information, the LR(k) method begins with a more elaborate set of items that already incorporates follow information. For example, given

$$\boxed{(3)\quad A \rightarrow \{\ \bullet\ E\,\}}$$

$$\boxed{(4)\quad B \rightarrow (\ \bullet\ E\,)}$$

the SLR method would assume that "}" or ")" could follow an E in any context. The LR(k) method carries into each state the relevant follow set. Thus, the table constructed by LR can have many more states than the table constructed by SLR.

LALR(k): is a compromise between SLR and LR. The table is the same size as SLR, but conflict resolution is sharper.

The methods described above are successful only for unambiguous grammars. Earley's algorithm (1, 16) can construct parses (and derivations) for ambiguous grammars. Note that LR parsing is more powerful that LL parsing.

What happens when LR(k) constructions fail?

If table construction reveals an inadequate state, one of the following must hold:

The grammar is ambiguous.

If the language is not itself inherently ambiguous, then perhaps the grammar can be modified to generate the same language, but unambiguously.

This is a task for human intelligence, as it's provably undecidable (*i.e.*, there is no mechanical process to decide) that a grammar is ambiguous.

A method that works well is to identify the inadequate states, and then work into and out of the state to generate a string that has more than one derivation. The conflicts (identified, for example, by YACC) are helpful in this process.

Underfueled table construction

1. **Generally, SLR is more powerful than LR(0); LALR is more powerful than SLR; LR is the most powerful (canonical) bottom-up parsing method.**
2. **Canonical LR parsers must form their reductions on top-of-stack. For some grammars (an example follows), no bounded amount of lookahead (bounded at table construction time) suffices to disambiguate some state.**

A good exercise is to attempt adding nested procedures into the ANSI C grammar. `foo(,,,...,) {` **becomes problematic: One can't tell whether** `foo` **is a procedure definition or invocation until the arbitrarily distant opening brace is seen.**

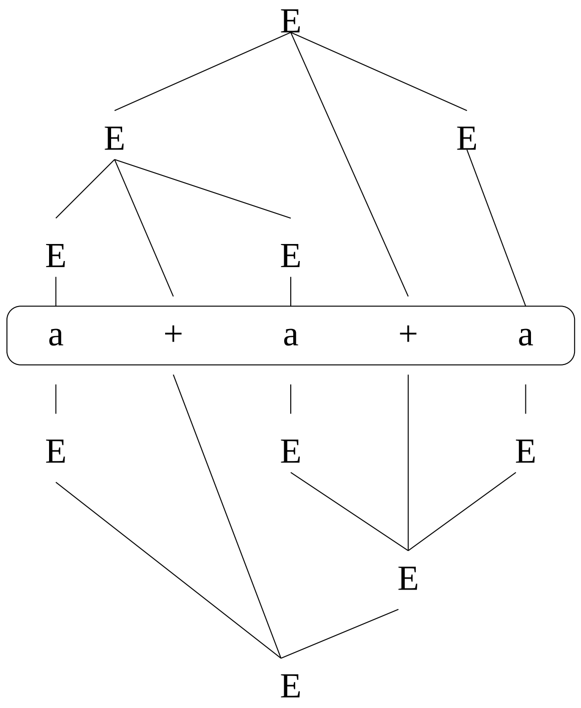
Identifying the cause of ambiguity

$$\begin{array}{rcl} E & \rightarrow & E + E \\ & | & a \end{array}$$

YACC **finds a shift/reduce conflict in the following state:**

(4)	$E \rightarrow$	$E \bullet + E$		Goto State	**3**
	$E \rightarrow$	$E + E \bullet$		Reduce by rule	$\boxed{1}$

Lining up the "dots" shows we can reach this state with the prefix ...E + E, and one rule shows how to continue this string to ...E + E + E.... We can now easily construct two parses: one assumes state 4 shifts (bottom), one assumes state 4 reduces (top).

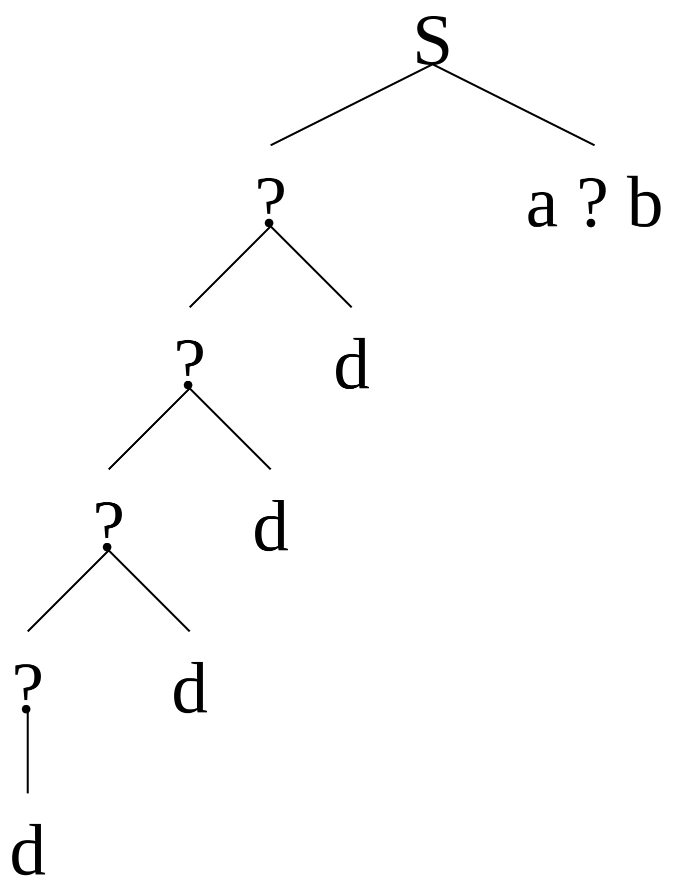


A grammar that is not LR(k) for any k

$$\begin{array}{lcl}
\mathbf{S} & \rightarrow & \mathbf{A\ a} \\
 & | & \mathbf{B\ b} \\
\mathbf{A} & \rightarrow & \mathbf{A\ d} \\
 & | & \mathbf{d} \\
\mathbf{B} & \rightarrow & \mathbf{B\ d} \\
 & | & \mathbf{d}
\end{array}$$

In the above grammar, a reduction must occur for the first "d" in the input, but the lookahead necessary for deciding whether to reduce A→ d or B→ d could be arbitrarily large.

If the right-hand sides of the first rules for A and B were reversed, then the grammar is LR(1), but the stack grows arbitrarily large at parse time.



Often the grammar can be modified to become LR(k), since this problem usually pertains to *how* the language is structured by the grammar.

Syntax + Semantics = Language

While the grammar for a language enforces syntactic constraints on accepted strings, some language issues are often postponed until after parsing.

For example, some language definitions contain rules that cannot be enforced by any context-free mechanism.

The most common examples involve some form of *type-checking*. Recall our expression grammar, a form of which appears in most programming language grammars. While the grammar allows an expression such as

$$a + b$$

most languages contain rules that restrict the types of a and b. For example, addition does not make sense if a is a character string and b is an array.

A grammar that accommodates type information would involve some *context*, and such grammars are difficult to design and expensive to process. Viable approaches to this problem involve some form of semantic processing, performed during or shortly after parsing:

Attribute grammars specify equations whose resolution essentially performs type checking.

Symbol tables are the most common solution. Type information is entered when identifiers are declared, so that expression types can be subsequently checked.

There is still the issue of whether type checking occurs in the same *pass* over the input as syntactic checking. Some languages forbid the kinds of "forward" declarations that would require extra passes for type checking.

Semantic processing

Also, there are often language constraints that are difficult or unwieldy to enforce syntactically.

For example, the ANSI C grammar essentially has a set of rules:

$$
\begin{array}{lcl}
\textbf{Declaration} & \rightarrow & \textbf{Qualifiers id} \\
\textbf{Qualifiers} & \rightarrow & \textbf{Qualifiers Qualifier} \\
 & | & \textbf{Qualifier} \\
\textbf{Qualifier} & \rightarrow & \textbf{int} \\
 & | & \textbf{float} \\
 & | & \textbf{static} \\
 & | & \textbf{extern} \\
 & | & \vdots
\end{array}
$$

While this grammar allows strings like

```
static int x
```

the grammar also admits strings such as

```
static int extern float x
```

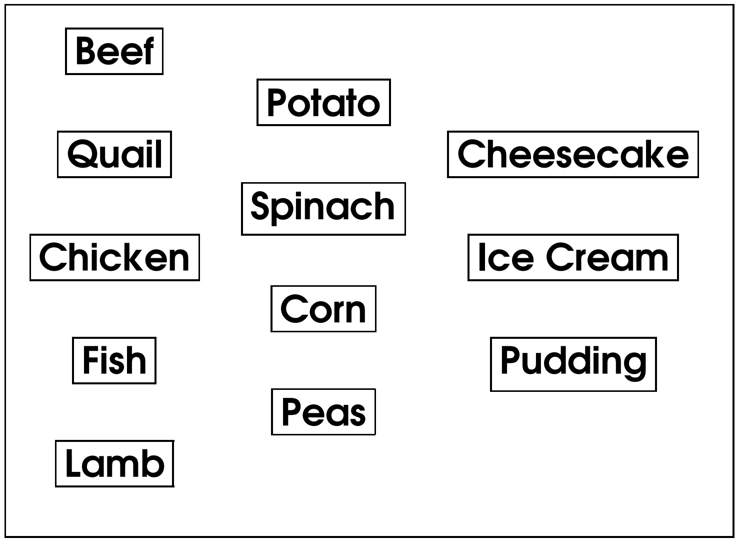
The language actually offers three kinds of type qualifiers. At most one from each category is allowed for any identifier.

The grammar could be transformed to enforce the kind and number of qualifiers that are allowed, but this would increase the size of the grammar.

Another example would be the *evaluation* of an expression. If we restricted the size of its terms, each expression could be syntactically evaluated by a huge grammar. Taken further, any programming language can be processed by a finite-state machine if the program size is bounded.

Ultimately, issues of taste and efficiently dictate how and where language issues are addressed.

Ordering from a Chinese menu



The rules for a "correctly" placed order are:

1. **At most one item may be selected from *any* column.**
2. **Some columns may be skipped.**
3. **At least one item must be chosen.**
4. **The items can be arbitrarily ordered.**

```
Order    →  Choices
Choices  →  Choice Choices
         |  Choice
Choice   →  ColA
         |  ColB
         |  ColC
ColA     →  BEEF
         |  CHICKEN
         |  QUAIL
         |  FISH
         |  LAMB
ColB     →  POTATO
         |  SPINACH
         |  CORN
         |  PEAS
ColC     →  CHEESECAKE
         |  PUDDING
         |  ICECREAM
```

The assignment is to rewrite the grammar to enforce the rules. This is exactly what's needed to enforce C's rules for declarations.

Solution

$$\begin{array}{lcl}
\text{Order} & \rightarrow & \text{Choices} \\
\text{Choices} & \rightarrow & \text{ColA} \\
 & | & \text{ColB} \\
 & | & \text{ColC} \\
 & | & \text{ColA ColB} \\
 & | & \text{ColB ColA} \\
 & | & \text{ColA ColC} \\
 & | & \text{ColC ColA} \\
 & | & \text{ColB ColC} \\
 & | & \text{ColC ColB} \\
 & | & \text{ColA ColB ColC} \\
 & | & \text{ColA ColC ColB} \\
 & | & \text{ColB ColA ColC} \\
 & | & \text{ColB ColC ColA} \\
 & | & \text{ColC ColA ColB} \\
 & | & \text{ColC ColB ColA}
\end{array}$$

$$\begin{array}{lcl}
\text{ColA} & \rightarrow & \text{BEEF} \\
 & | & \text{CHICKEN} \\
 & | & \text{QUAIL} \\
 & | & \text{FISH} \\
 & | & \text{LAMB} \\
\text{ColB} & \rightarrow & \text{POTATO} \\
 & | & \text{SPINACH} \\
 & | & \text{CORN} \\
 & | & \text{PEAS} \\
\text{ColC} & \rightarrow & \text{CHEESECAKE} \\
 & | & \text{ICECREAM} \\
 & | & \text{PUDDING}
\end{array}$$

While some factoring of this grammar is possible, this example illustrates the tradeoff between grammar size and specificity of the parse.

Symbol tables

The symbol table tracks symbols and their types, where type information could be any property of a symbol relevant to subsequent activity in the compiler.

```
static char *a[5];
```

is an array of 5 pointers to characters.

Such information typically includes

- **the basic type of a variable (ptr, int, char, float, struct, etc.);**
- **structure layout, pointer specifics, array information;**
- **initialization values;**
- **scope information.**

I provide the following symbol table access functions:

IncrNestLevel(): increase the nest level by one.

DecrNestLevel(): decrease the nest level by one.

EnterSymbol(M,name): enters the string **name** as a symbol of type **M** at the current nest level.

RetrieveSymbol(name): returns a pointer to the currently active declaration of **name**. If **name** is not active, an error message is produced and the parse is aborted.

ExistsSymbol(name): operates like **RetrieveSymbol()**, but instead of aborting, a NULL pointer is returned if **name** isn't active.

I provide extra credit for those who implement their own, hash-based symbol table manager.

Symbol tables

Essential information

1. **Names;**
2. **Scope information;**
3. **Type information;**
4. **Storage specifics.**

Issues

1. **Programs typically contain a mix of very long and very short names (`i` *vs.* `WindowMaxAccelScreenMouse()`).**
2. **Type checking and code generation do not require access to all scopes at all times. Typically, access is required only to the current scope and its outer scopes. Even then, programs use the current and outermost scopes most frequently.**

There are two popular methods of establishing symbol tables:

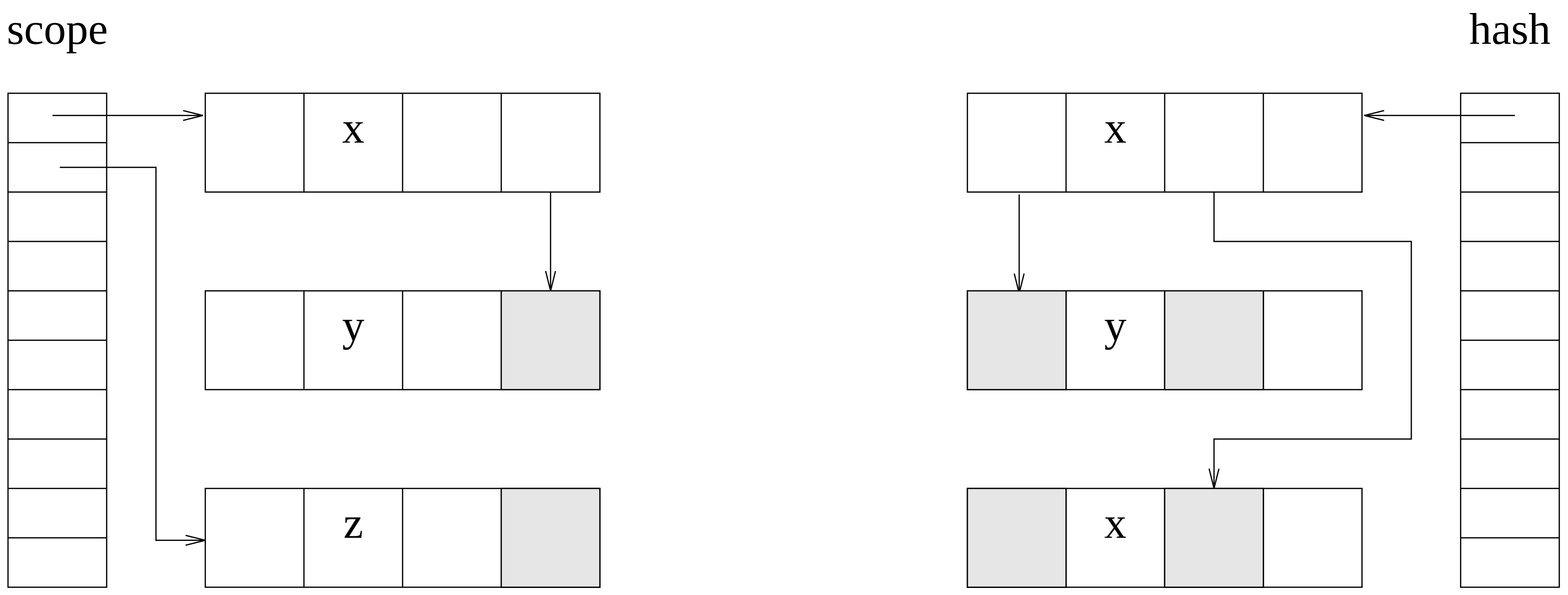
1. Make a separate pass over the program to create the symbol table;
2. Build the symbol table as you parse.

Given that one typically creates an abstract syntax tree anyway, it seems wise to defer symbol table creation to a separate pass. On the other hand, restructuring the grammar to simplify symbol table creation is a good exercise, and it is necessary for a one-pass compiler.

Symbol table organization

Each cell

Same hash (H)	Actual symbol info	Same ID (V)	Same scope (L)



The above scheme implements a stack for each variable v, where top-of-stack is the currently active instance of v. Let $f(v)$ be the hash index for variable v:

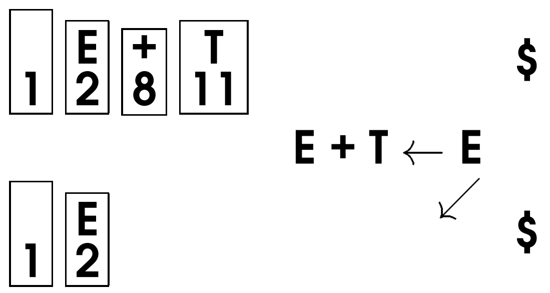
Entering a scope: each variable v is pushed onto the stack headed by its (chained) hash index $f(v)$.

Leaving a scope k: each variable linked from scope k is popped off its stack.

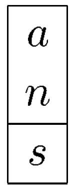
Lookup: use $f(v)$, with chaining via H, to locate the named variable.

Semantic processing at parse time

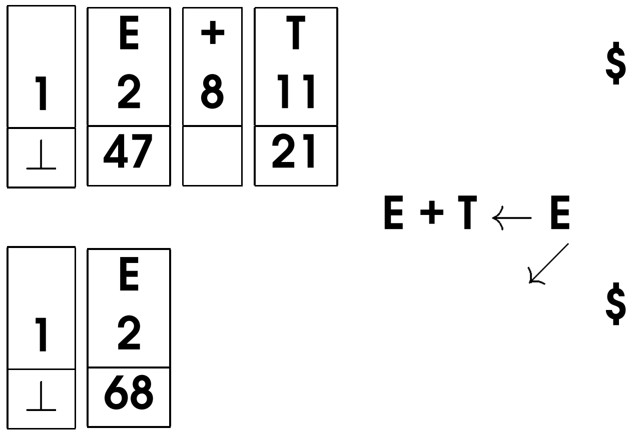
Recall how an LR parser uses a stack to apply reductions:



Our parse stack previously consisted of $\boxed{\begin{smallmatrix}a\\n\end{smallmatrix}}$ **, where** a **is a grammar symbol and** n **is a parse state. We now augment the stack to contain semantic information:**



We can use this semantic stack to *synthesize* **information during the parse. In our example, when E+T is reduced, we could add together the values associated with E and T, pushing the sum on the stack along with the E replacing the E+T:**



In YACC, the grammar file can specify a union of types for the semantic stack, so that information of any form can be synthesized during the parse.

Synthesized attributes

With YACC, a segment of C code can be associated with each production. Given a rule

$$\mathbf{A} \rightarrow a_1\, a_2 \ldots a_k$$

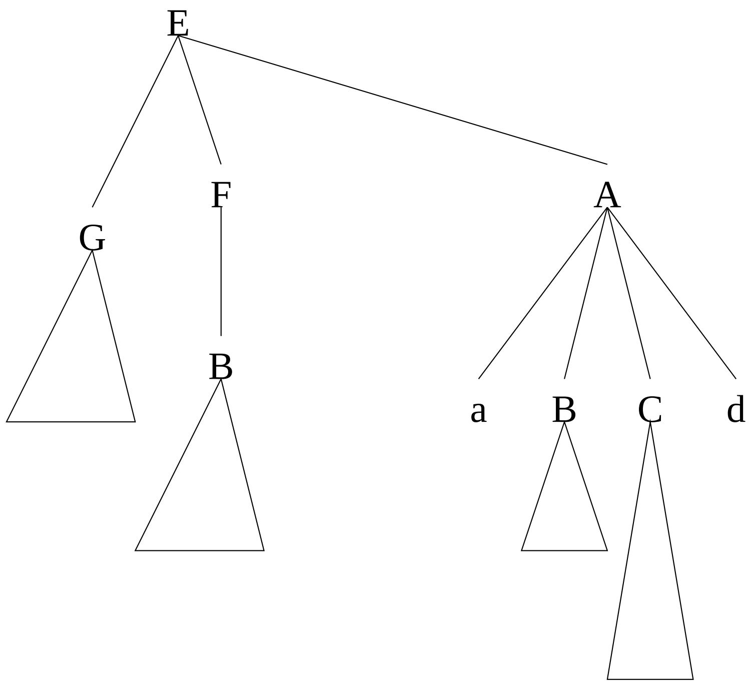
the segment of C code can refer to the semantic stack values of symbols as follows:

Rule Symbol	Semantic Value
a_1	**\$1**
a_2	**\$2**
⋮	
a_k	**\$k**
A	**\$\$**

so that a typical rule looks like

$$\mathbf{A} \rightarrow a_1\, a_2 \ldots a_k$$

```
{$$ = $3 + f($2);}
```



When "a B C d" is reduced to A, information previously contributed to B and C can be incorporated into the information synthesized for A. Without global storage, information computed at a tree node X is a pure function of information computed at X's descendents.

More generally, one can reference any value still on the stack. In our example,this include information associated with F and G. Such grammars are called *L-attributed*.

Evaluating infix expressions

S $\rightarrow$ **E $**

`{printf("Answer is %d\n",$1);}`

E $\rightarrow$ **E + T**

`{$$ = $1 + $3;}`

| **T**

`{$$ = $1;}`

T $\rightarrow$ **T** $*$ **F**

`{$$ = $1 * $3;}`

| **F**

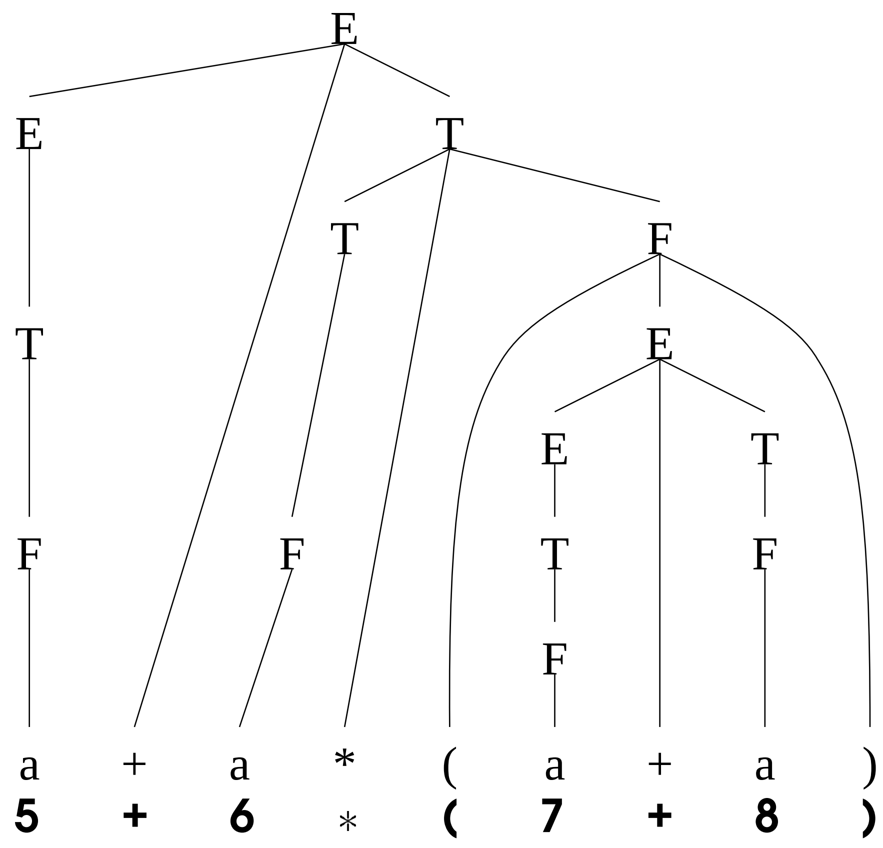
`{$$ = $1;}`

F $\rightarrow$ **(E)**

`{$$ = $2;}`

| **const**

`{$$ = $1;}`



Notice how the unit productions ($A \rightarrow B$) lead to simple copying of values up the parse tree. While these rules participate in disambiguating the grammar, they are not always conducive to semantic processing.

Another example

Num $\rightarrow$ **D $**

`{printf("Answer:  %d\n",$1);}`

D $\rightarrow$ **D d**

`{$$ = (10 × $1) + $2;}`

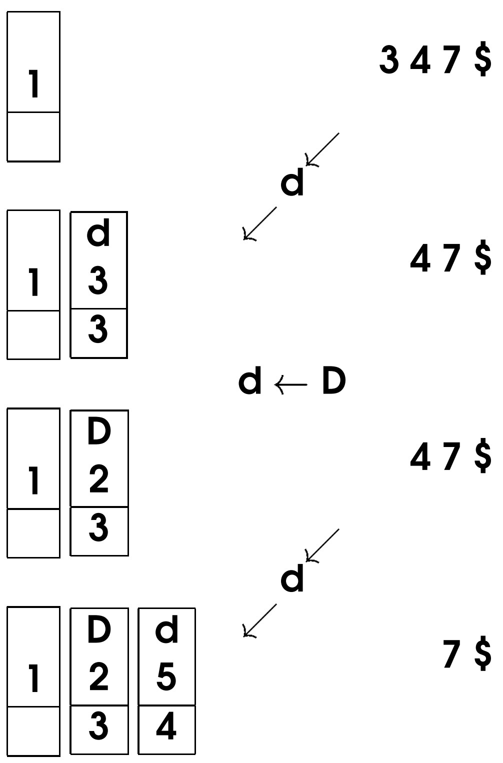
| **d**

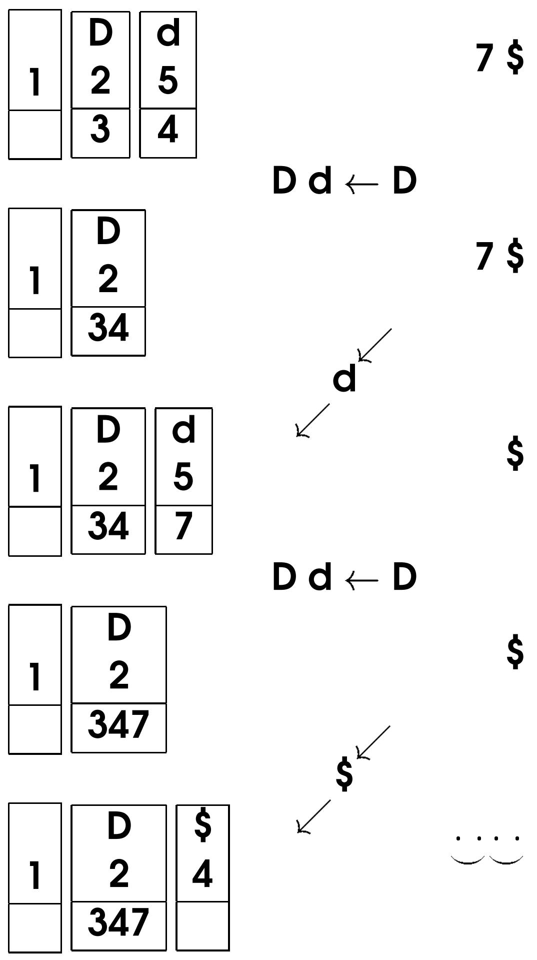
`{$$ = $1;}`

State	d	$	D
1	Goto State 3		Goto State 2
2	Goto State 5	Goto State 4	
3	Reduce by rule $\boxed{1}$	·	Reduce by rule $\boxed{1}$
4	☺	·	☺
5	Reduce by rule $\boxed{2}$	·	Reduce by rule $\boxed{2}$

Note: grammar is LR(0)

Applied to the string "347$"





Grammars and semantic processing

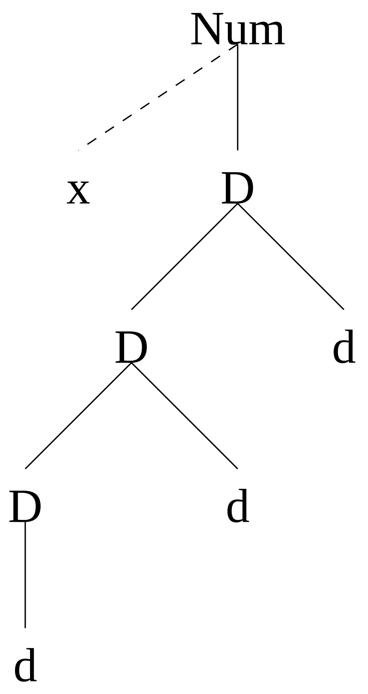
There are usually many unambiguous grammars that generate a given programming language. In planning for semantic processing, it is often convenient to rewrite the grammar so that reductions and stack activity are conducive to the required actions.

Consider the grammar:

$$\begin{array}{lcl} \textbf{Num} & \rightarrow & \textbf{x D} \\ & \mid & \textbf{D} \\ \textbf{D} & \rightarrow & \textbf{D d} \\ & \mid & \textbf{d} \end{array}$$

Interpretation: a string of digits represents a base-10 number, unless the string is preceded by an ‘x’, in which case the string represents a base-8 number.

String	Number
3 4 7	347
x 3 4 7	231



We could compute the number by passing the list of digits up the tree, forming the answer at Num. We would prefer to compute the number *as we reduce the digits,* but this grammar’s parse trees have the base information in the wrong place.

Rewriting the grammar

Num $\rightarrow$ **x OctD $**
```
{printf("Answer:  %d\n",$2)}
```
| **DecD $**
```
{printf("Answer:  %d\n",$1)}
```
DecD $\rightarrow$ **DecD d**

{$$ = $(10 \times \$1) + \$2$;}

| **d**

{$$ = $1;}

OctD $\rightarrow$ **OctD d**

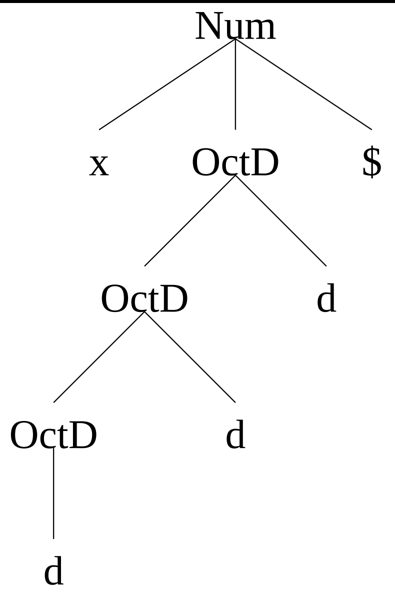
{$$ = $(8 \times \$1) + \$2$;}

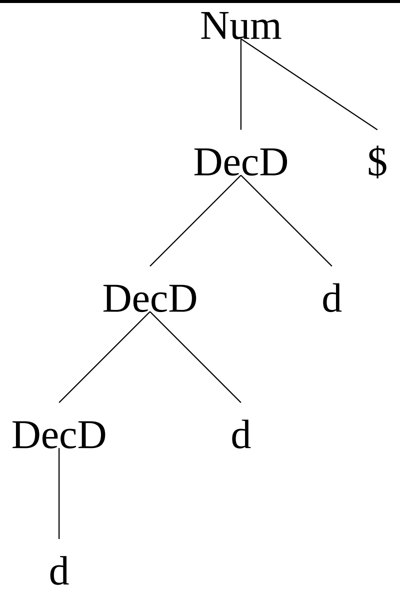
| **d**

{$$ = $1;}

State	d	x	$	DecD	OctD
1	Goto State **4**	Goto State **2**			
2	Goto State **6**				Goto State **5**
3	Goto State **8**		Goto State **7**		
4	Reduce by rule $\boxed{4}$	· · ·	· · ·	· · ·	Reduce by rule $\boxed{4}$
5	Goto State **10**		Goto State **9**		
6	Reduce by rule $\boxed{6}$	· · ·	· · ·	· · ·	Reduce by rule $\boxed{6}$
7	☺	· · ·	· · ·	· · ·	☺
8	Reduce by rule $\boxed{3}$	· · ·	· · ·	· · ·	Reduce by rule $\boxed{3}$
9	☺	· · ·	· · ·	· · ·	☺
10	Reduce by rule $\boxed{5}$	· · ·	· · ·	· · ·	Reduce by rule $\boxed{5}$

Note: grammar is LR(0)





Another change!

Suppose we want the base itself to be part of the input:

String	Number
3 4 7	347
x 8 3 4 7	231
x 9 3 4 7	286

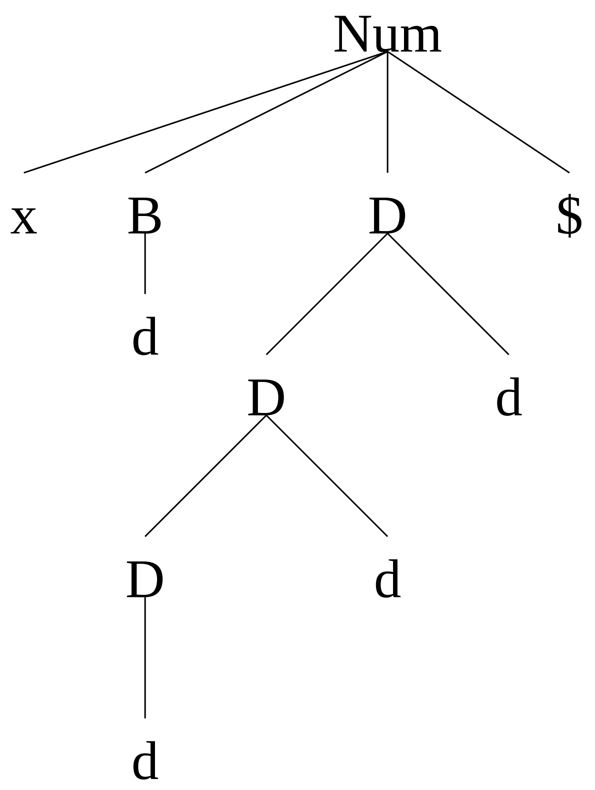
One possibility is to use a global variable:

Num $\rightarrow$ **x B D $**
`{printf("Answer:  %d\n",$3);}`
$|$ **Skip D $**
`{printf("Answer:  %d\n",$2);}`

B $\rightarrow$ **d**
`{Base = $1;}`

Skip $\rightarrow$ λ
`{Base = 10;}`

D $\rightarrow$ **D d**
{`$$ = `(`Base` $\times$ `$1`) + `$2;`}
$|$ **d**
`{$$ = $1;}`

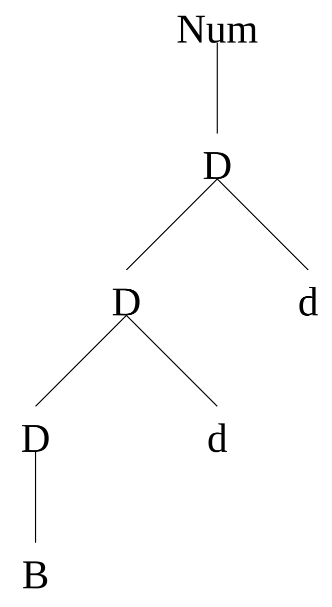


Note that the reduction B $\rightarrow$ d is necessary to set the global variable. In the LR parse, this is the first reduction, so the base will indeed be set when the first D $\rightarrow$ D d rule is applied. But global variables are not very clean, especially if constructs could be nested so that global variables get overwritten.

Arriving at a good grammar

Let's engineer the tree we would like to see, and then construct the appropriate grammar. In the tree shown to the right, it's possible to synthesize the base up the tree.

At each reduction, we could know how to compute the new value to pass up the tree.



Num $\rightarrow$ D $

    {printf("Answer: %d\n",$1.value);}

D $\rightarrow$ D d

    {$$.value = ($1.base × $1.value) + $2;
    $$.base = $1.base;}

  | B

    {$$.base = $1;}

B $\rightarrow$ x d

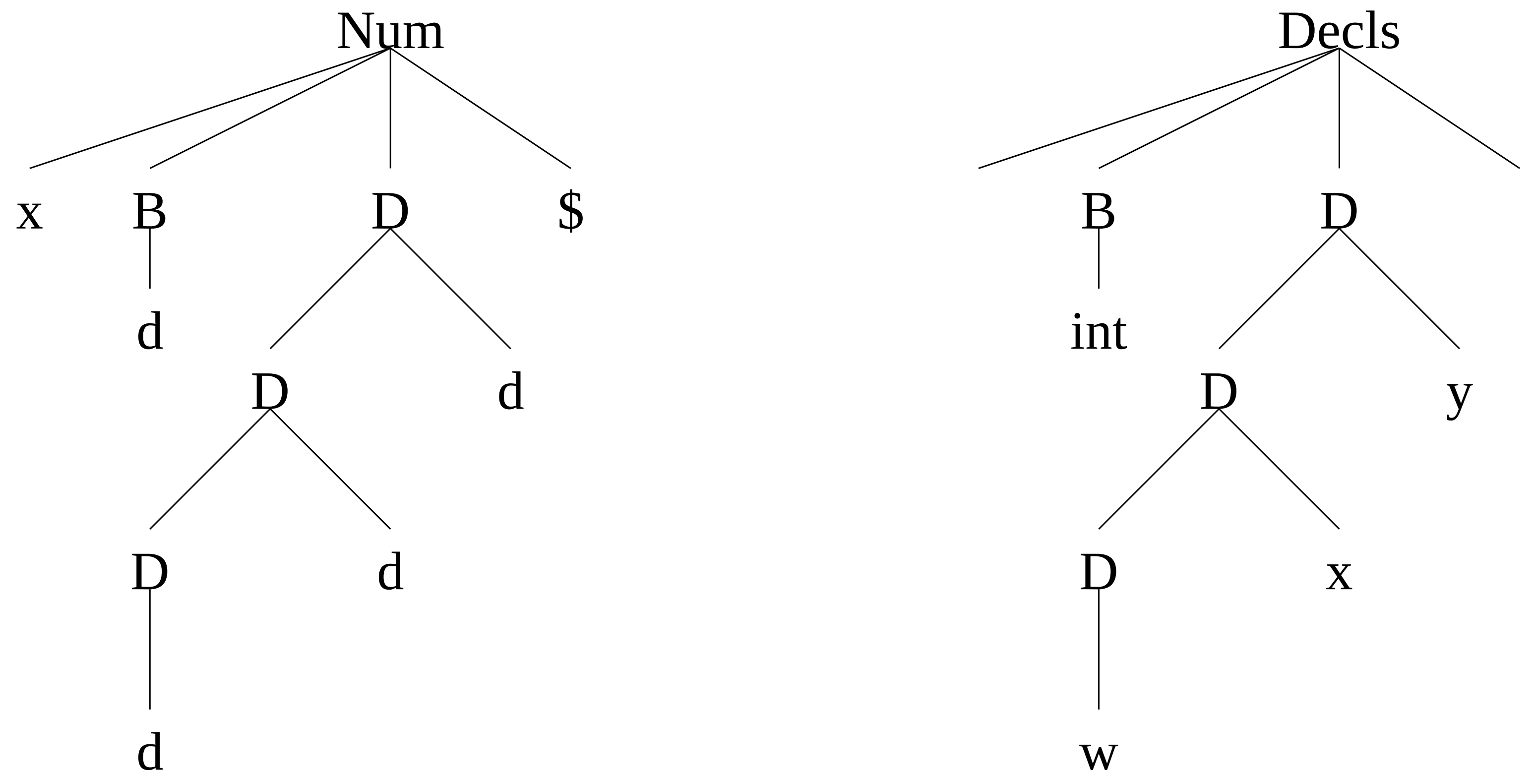
    {$$ = $2;}

  | λ

    {$$ = 10;}

Back to declarations

The running example of converting a string of digits to a number is actually an abstraction of processing variable declarations in C.



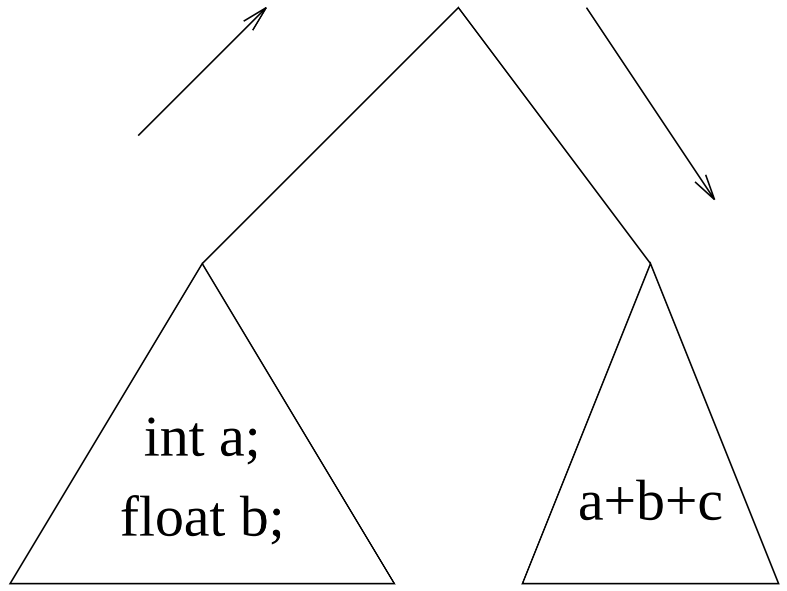
We would like to enter the variables in the symbol table, along with their types, as we parse the input. Rewriting the ANSI C grammar to accomplish this is a good exercise.

Note that PASCAL has its type information at the end, and so a right recursive rule can similarly accommodate that form of syntax.

Back to symbol tables

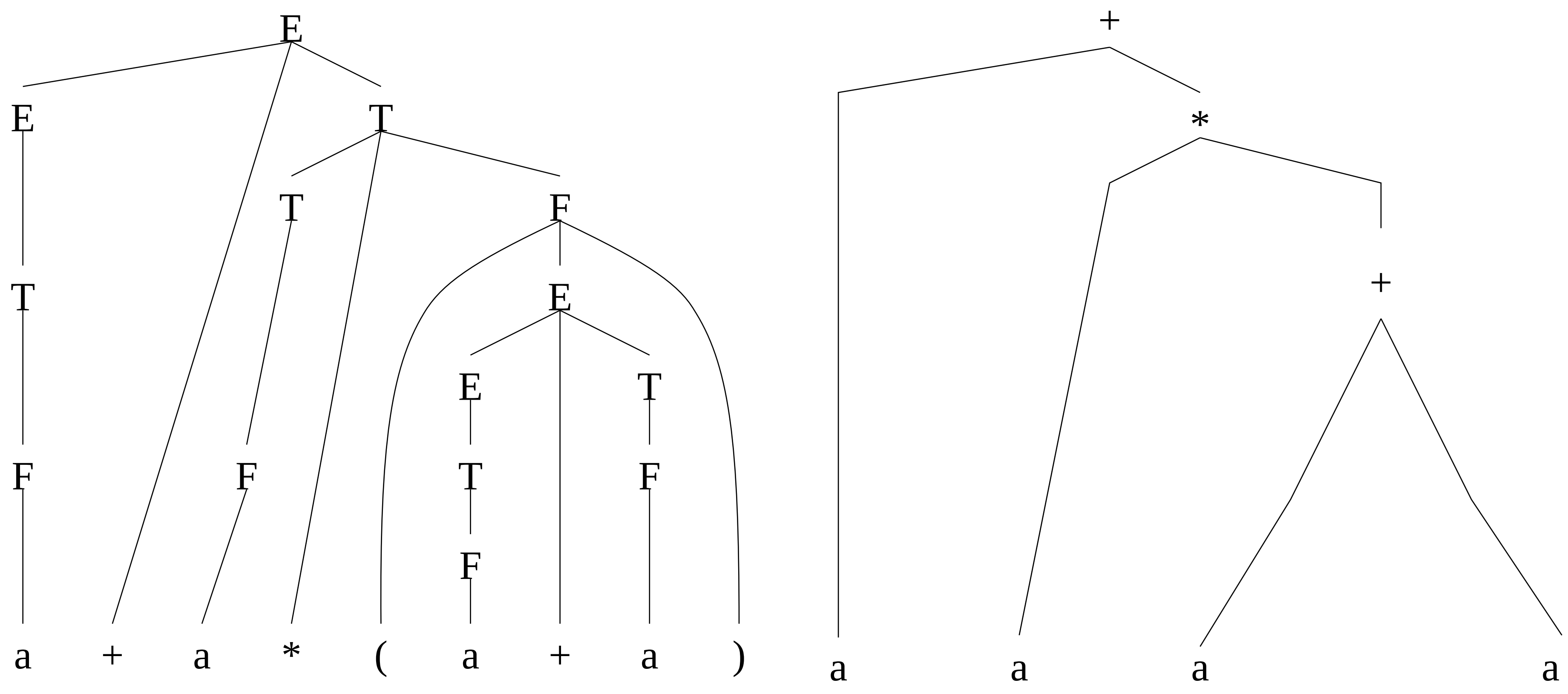
An *attributed grammar* **allows semantic equations whose terms depend on synthesized and** *inherited* **attributes. A classical use of attribute grammar systems is for the synthesis and use of type information.**

As the declarations are parsed, a "symbol table" is synthesized up the parse tree. While processing the code of a procedure, this symbol and those from outer scopes are available as an inherited attribute.



While attribute grammars offer a clean mechanism for expressing semantics, such systems are usually slower than those involving only synthesized attributes, and one must still get the equations "right". The Cornell Program Synthesizer is a popular and robust system for developing compilers based on attribute grammars (41, 31, 32).

Abstract syntax trees (ASTs)



The AST eliminates the scaffolding introduced to render the grammar unambiguous. Items such as temporary variables can be introduced into the AST to simplify subsequent activity (optimization, code generation).

Creating an AST

We can easily add actions to the grammar to create AST nodes and properly link these nodes to form the AST.

$$\begin{array}{lll}
\mathbf{S} & \rightarrow & \mathbf{E}\ \$ \\
\mathbf{E} & \rightarrow & \mathbf{E} + \mathbf{T} \\
 & & \{\texttt{\$\$ = MakeBinTree(PLUS,\$1,\$3);}\} \\
 & | & \mathbf{T} \\
 & & \{\texttt{\$\$ = \$1;}\} \\
\mathbf{T} & \rightarrow & \mathbf{T} * \mathbf{F} \\
 & & \{\texttt{\$\$ = MakeBinTree(TIMES,\$1,\$3);}\} \\
 & | & \mathbf{F} \\
 & & \{\texttt{\$\$ = \$1;}\} \\
\mathbf{F} & \rightarrow & (\ \mathbf{E}\ ) \\
 & & \{\texttt{\$\$ = \$2;}\} \\
 & | & \mathbf{const} \\
 & & \{\texttt{\$\$ = MakeConst(\$1);}\} \\
 & | & \mathbf{id} \\
 & & \{\texttt{\$\$ = MakeSymb(\$1);}\}
\end{array}$$

Free of clutter, the resulting tree can then be traversed to instantiate symbol tables, perform type checking, optimize the program, and generate code.

AST routines

```
typedef struct _TreeNode {
   struct {
     int linenumber;
     int colnumber;
   } sourceinfo;
   NodeInfo info;
   struct _TreeNode *child;
   struct _TreeNode *sibling;
   struct _TreeNode *head;
   struct _TreeNode *parent;
   struct _TreeNode *leftsib;
} TreeNode;
```

NodeInfo is a *union* of tree node information: symbol table pointers, integer values, operator types, etc.

$MakeFamily(parent, sibs)$: **adopts $sibs$ into the $parent$'s family, returning the parent.**

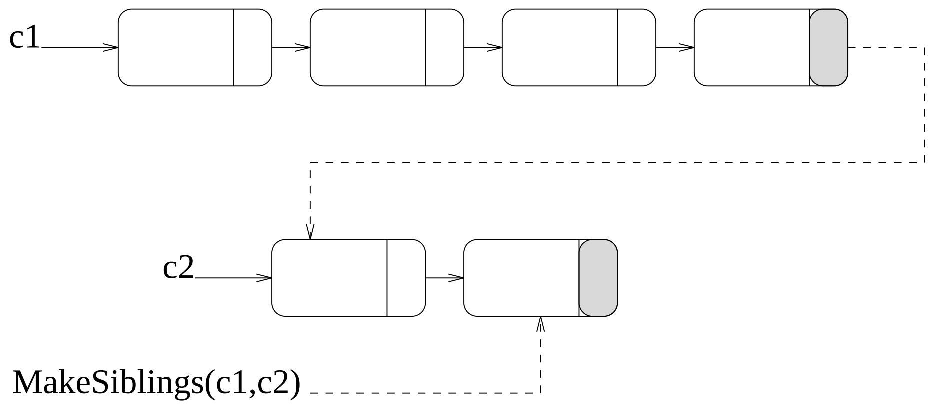
$MakeSiblings(c1, c2)$: **units siblings $c1$ and $c2$, returning the end of the resulting list (shown below).**

$MakeOperatorNode(opnum)$: **creates an operator node, where $opnum$ is the "name" of a "token".**

$MakeIntegerNode(intval)$: **creates an integer node with value $intval$.**

$MakeStringNode(str)$: **creates a string node with value str.**

$MakeSymbolNode(sym)$: **creates a symbol reference node to sym.**



Using the AST routines

Num $\rightarrow$ **D $**

```
        {$$ = $1 → head;}
```

D $\rightarrow$ **D d**

```
        {$$ = MakeSiblings($1,
                 MakeIntegerNode($2));}
```

$|$ **B**

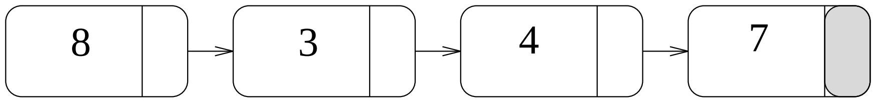
```
        {$$ = MakeIntegerNode($1);}
```

B $\rightarrow$ **x d**

```
        {$$ = $2;}
```

$|$ λ

```
        {$$ = 10;}
```



The above list is created by the actions shown to the left. The first number in the list is the base, and the subsequent numbers are the digits as parsed from left to right.

Example AST

```
int a1;
extern int a2;

int factorial(X)
int X;
{
  int Y;

  Y = X;
      if (Y > 0) [ Y*factorial(X-1); ];
               else [1;];
}

void main() {
   int i;
   a1 = factorial(i=5);
   a2 = factorial(3);
}
```

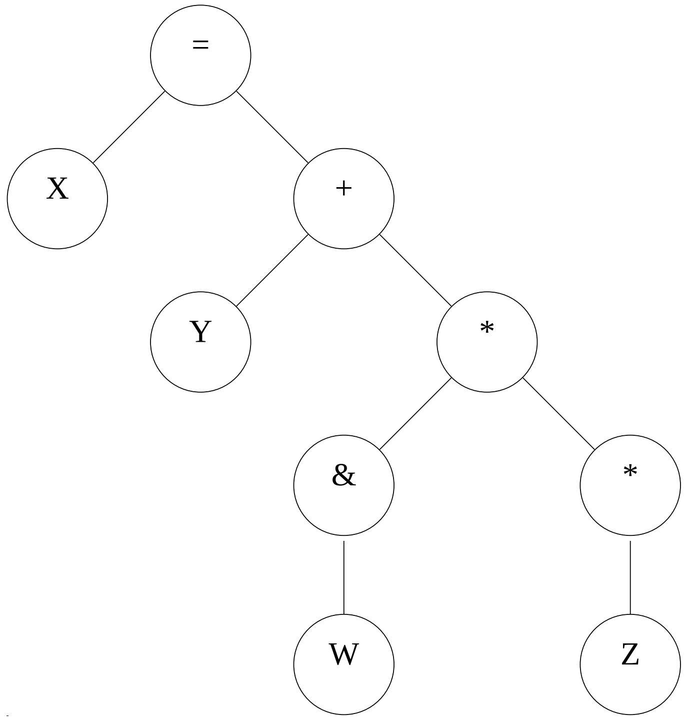
The AST is shown to the right, with indentation reflecting tree depth.
Note the regular structure:

- functions and inline procedures are represented similarly.
- an if-then structure is represented as an if-then-else with trivial "else" code.

```
Operator PROGRAM
  Operator FORMALS
  Operator SDCLS
    Ref Symbol001(02)  int   *0    [0] auto         : a1
    Ref Symbol002(02)  int   *0    [0] extern       : a2
  Operator FDCLS
    Ref Symbol003(02)  int   *0 () [0] auto         : factorial
      Operator FORMALS
        Ref Symbol004(03)  int   *0    [0] auto         : X
      Operator SDCLS
        Ref Symbol005(04)  int   *0    [0] auto         : Y
      Operator FDCLS
      Operator EXPRBLOCK
        Operator OTHEREXPRS
          Operator ASSIGN
            Ref Symbol005(04)  int   *0    [0] auto         : Y
            Ref Symbol004(03)  int   *0    [0] auto         : X
        Operator LASTEXPR
          Operator IF
            Operator GT_OP
              Ref Symbol005(04)  int   *0    [0] auto         : Y
              Integer 0
            Operator INLINEPROC
              Operator FORMALS
              Operator SDCLS
              Operator FDCLS
              Operator EXPRBLOCK
                Operator OTHEREXPRS
                Operator LASTEXPR
                  Operator TIMES
                    Ref Symbol005(04)  int   *0    [0] auto         : Y
                    Operator INVOKE
                      Ref Symbol003(02)  int   *0 () [0] auto         : factorial
                      Operator ARGS
                        Operator MINUS
                          Ref Symbol004(03)  int   *0    [0] auto         : X
                          Integer 1
            Operator INLINEPROC
              Operator FORMALS
              Operator SDCLS
              Operator FDCLS
              Operator EXPRBLOCK
                Operator OTHEREXPRS
                Operator LASTEXPR
                  Integer 1
    Ref Symbol006(02)  void  *0 () [0] auto         : main
      Operator FORMALS
      Operator SDCLS
        Ref Symbol007(04)  int   *0    [0] auto         : i
      Operator FDCLS
      Operator EXPRBLOCK
        Operator OTHEREXPRS
          Operator ASSIGN
            Ref Symbol001(02)  int   *0    [0] auto         : a1
            Operator INVOKE
              Ref Symbol003(02)  int   *0 () [0] auto         : factorial
              Operator ARGS
                Operator ASSIGN
                  Ref Symbol007(04)  int   *0    [0] auto         : i
                  Integer 5
        Operator LASTEXPR
          Operator ASSIGN
            Ref Symbol002(02)  int   *0    [0] extern       : a2
            Operator INVOKE
              Ref Symbol003(02)  int   *0 () [0] auto         : factorial
              Operator ARGS
                Integer 3
  Operator EXPRBLOCK
    Operator OTHEREXPRS
    Operator LASTEXPR
```

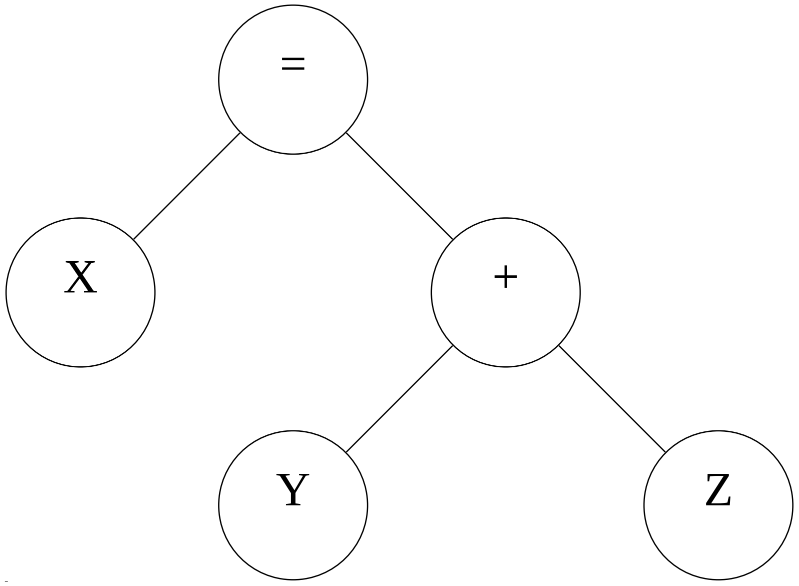
Type checking

L *vs.* R values



The actual meaning of the identifier is dependent on its context.

Type compatibility



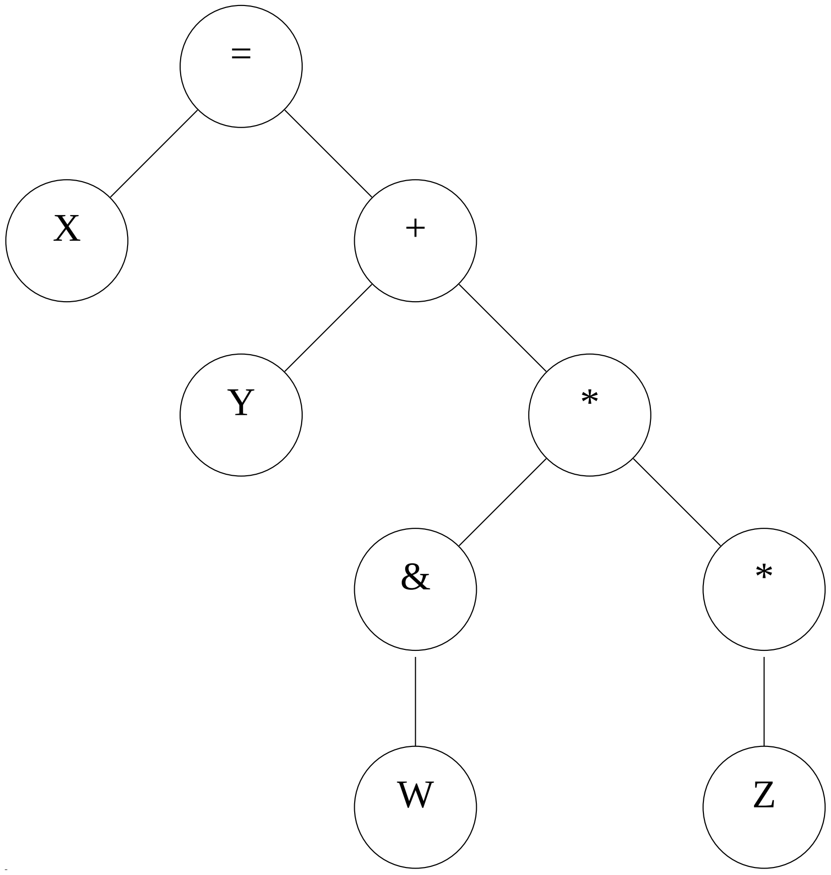
The meaning of $+$ in the above program depends on the types of Y, Z, and X. In languages that allow operator overloading, even the meaning of $+$ becomes suspect.

Notice the dual role of the operator $*$ (in the C language), which is nicely disambiguated using the proper grammar.

Left and right values of identifiers

Named for their interpretation with respect to "=", the *left* value of an identifier is its location while the *right* value is the contents of the identifier.

L *vs.* R values



The positioning of identifiers with respect to various operators in C indicates which value is desired:

X=	**The storage location of X**
=Y	**The value stored at Y**
$*$ Z	**The value at Z, treated as a storage location**
& W	**The address of W, treated as a value**

C left and right values

Form	Expects	Produces
a=b	LV(a), RV(b)	RV
$*$ c	RV(c)	LV
&d	LV(d)	RV

$$
\begin{array}{lcl}
S & \rightarrow & L = R \\
 & | & R \\
L & \rightarrow & id \\
 & | & * R \\
R & \rightarrow & L \\
 & | & \& L \\
 & | & int
\end{array}
$$

This grammar produces structures where the interpretation of left and right values is clear.

Moreover, the rule R→L is applied when a left value "becomes" a right value through dereferencing.

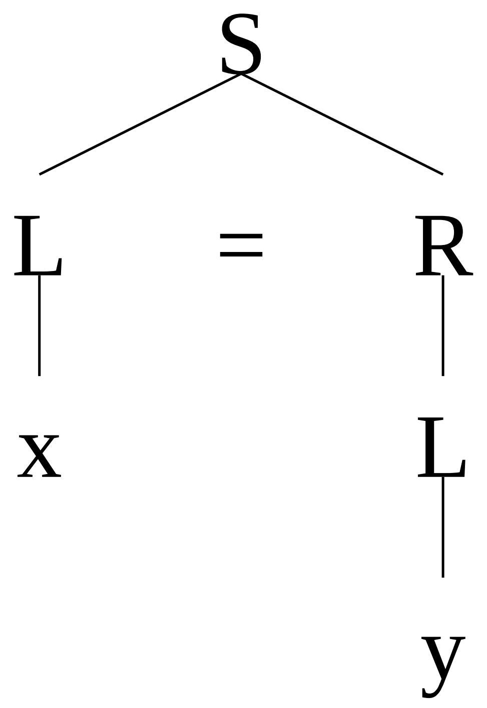
The grammar correctly precludes strings like "3=x" and "&z=y".

Table construction for this grammar fails for SLR because "$=$" can follow an R.

But there's no sentential form that begins "R="; the R must be preceded by an $*$ as in "$*$R=". The LR(1) construction can create a suitable parse table. The grammar is also LALR(1) (YACC can handle this grammar).

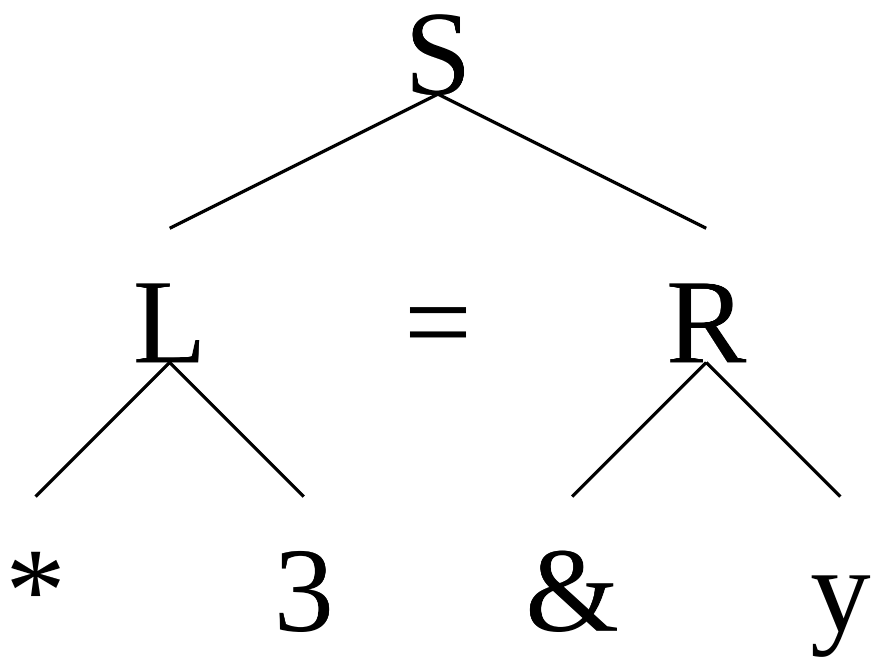
Examples

x=y



Push x
Push y
Fetch
Store

$*$3=&y

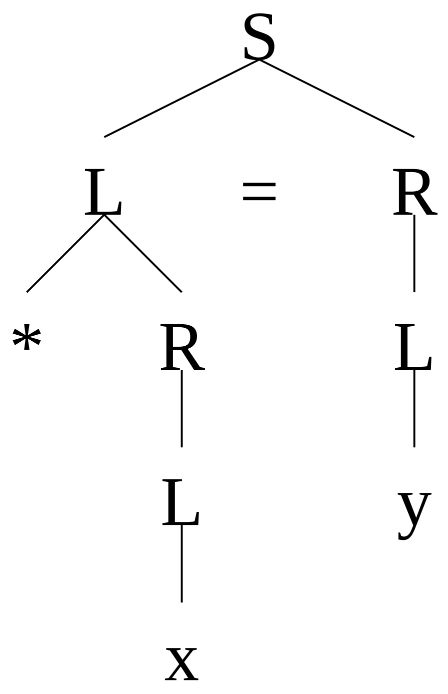


Push 3
Push y
Store

The $*$ and & have no effect on code generation: they merely change the type of an expression. The language design is biased towards the most prevalent "x=y" form.

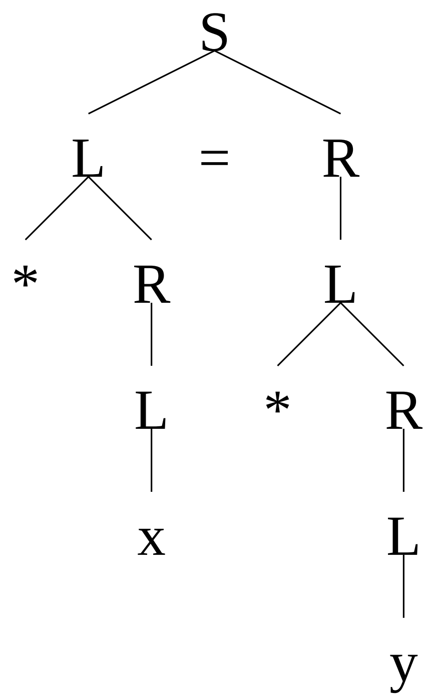
Examples

$*$x=y



Push x
Fetch
Push y
Fetch
Store

$*$x=$*$y



Push x
Fetch
Push y
Fetch
Fetch
Store

More on left and right values

Unfortunately, the syntactic rules for C do not allow a grammar-based approach to left and right value disambiguation. However, the rules related to $=$, &, and $*$ can be applied just as easily to the parse tree, using attribute grammars or an additional (bottom-up) pass over the tree.

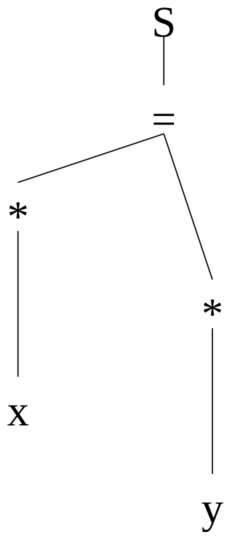
SetV($node, kind$): asserts the left or right valuedness of $node$.

ExpectLV($node$): expects that $node$ is a left value.

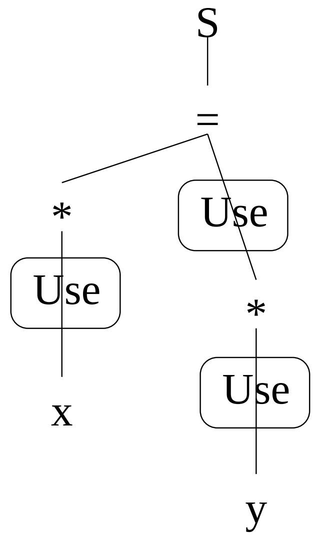
ExpectRV($node$): expects that $node$ is a right value.

Convert($node, how$): attempts to convert $node$ into a left or right value.

As our grammar indicates, the only conversion that makes sense is a left to right value conversion, which is basically a dereference. In a call-by-reference language, however, a right value could become a left value by introducing a temporary. Show below are the parse trees before and after the extra bottom-up pass.



Form	Expects	Produces
a=b	LV(a), RV(b)	RV
$*$ c	RV(c)	LV
&d	LV(d)	RV



Data types and compatible operations

The type checking phase of a compiler is traditionally responsible for establishing the semantic well-formedness of operations and data. Where language standards allow flexibility (some would say sloppiness) with respect to type consistency, compilers are charged with introducing implicit or explicit conversion operations to allow operations on otherwise unsuitable data.

Most languages offer a host of basic types, which are usually (though not always) supported by target instruction sets:

- **integers;**
- **floating point;**
- **Boolean-valued $\{$ true, false $\}$;**
- **character.**

Most languages also allow the introduction of new types based on old ones:

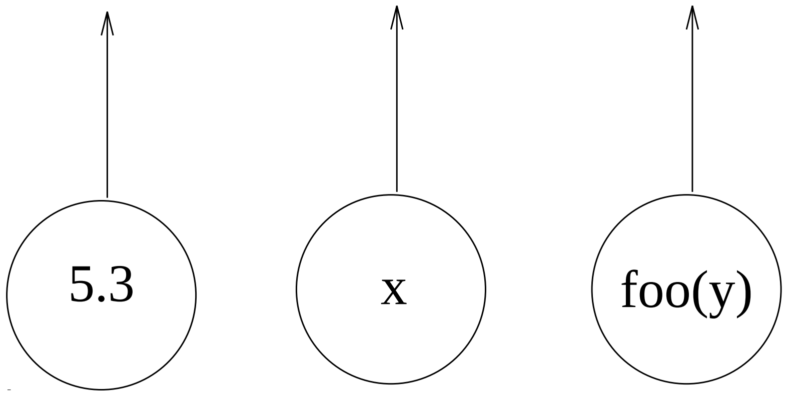
- **tuples (records, structs);**
- **maps (functions, arrays);**
- **sets.**

Proponents of *strongly-typed* languages, where data and operations must adhere rigidly to type consistency, claim that when soundly checked at compile-time, their programs are less likely to contain bugs.

Type checking

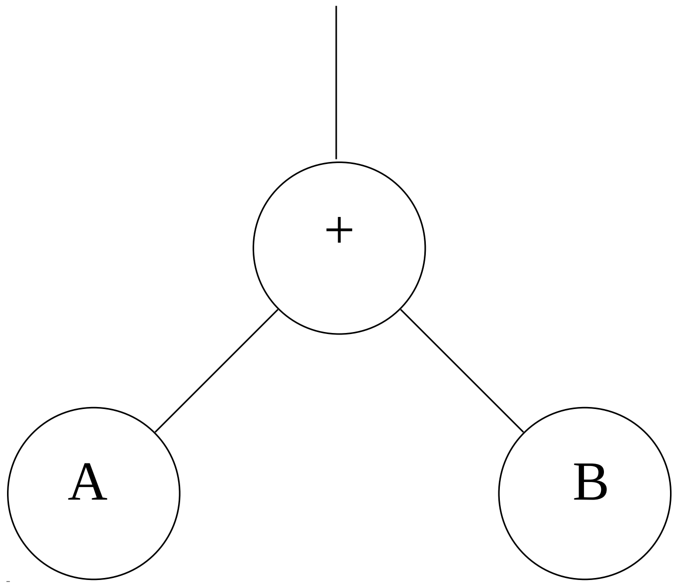
As with left and right value determination, type checking can be performed as a bottom-up pass over the parse (or abstract syntax) tree.

A straightforward (*i.e.*, highly localized) scheme operates as follows. At the tree's leaves are found the atomic elements such as constants, identifiers, and function calls. Each of these asserts its type, based on syntactic ("x" *vs.* 'x') or contextual (declared) information.



At each internal node X

1. **the subtrees of X are checked for type compatibility: this depends on the operation contained in X;**
2. **conversion operations are inserted as necessary;**
3. **the type of X is determined.**



Simple type checking

$+$	int	float	char
int	int	float	int
float	float	float	float
char	int	float	int

$=$	int	float	char
int	int	int	int
float	float	float	float
char	char	char	char

The arithmetic operators tend to find the grandest type suitable for performing the operation, while the assignment operators insist that the assigned value matches the type of its destination.

Name *vs.* structural type equivalence

```
typedef int t
int x
t y
⋮
x = (int) y
```

```
enum { a=1, b=2, c=3} foo;
⋮
foo = foo + 1;
M[(int) foo] = ‘x’;
```

The C language has *cast* operations, that assert the type of an expression. But conversions can also occur in uncast expressions, which can lead to confusion.

Are the above casts necessary? It depends on whether we regard type equivalence as a *structural* property or as a property of the name used in the declaration. In the above examples, `x`, `y`, and `foo` are all structurally represented as an integer.

The use of $+$ on `foo` could also be problematic, if an `enum` data type cannot be the target of $+$.

Pointers are an interesting example, since they are all structurally the same. Good language design and programming practice suggest distinguishing between pointers to different types.

C castigates those who fail to cast between pointers of different types.

Representing types [33]

The lineage of a type can be represented without reference to specific type names. A bit-vector representation is convenient for construction and for comparison:

Each of the types extenders is assigned a bit pattern from k bits:

Type	Pattern
ptr	01
array	10
func	11

So that a pointer to type X is represented as

01X

Note that this scheme does not track array index types or function parameter types.

Wisely leaving one pattern free, we now assign the base types:

Type	Pattern
void	0000
char	0001
int	0010
float	0011

Declaration	Type representation
`int x`	0010
`int **x[]`	01 01 10 0010
`char (*(*x())[])()`	11 01 10 01 11 0001

The last entry is a function that returns a pointer to an array of pointers to functions that return characters.

Runtime storage management

Most block-structured languages require manipulation of runtime structures to maintain efficient access to appropriate data and machine resources. For our purposes, a procedure is either a named function or an inline (parameterless) block.

Let's examine the activity normally associated with invoking a procedure P:

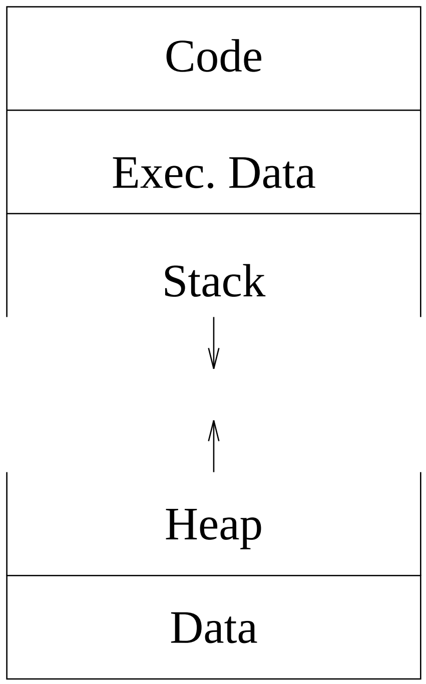
1. **Some machine state might be saved: general registers, vector registers, condition codes, interrupt masks, etc.**
2. **Access must be established to P's local variables and compiler-generated temporaries.**
3. **Access must be established to outer scope variables (but not for** C**).**
4. **The caller of P must be recorded so that P can return when done.**
5. **Parameters might be received prior to executing P.**
6. **A return value might be prepared prior to returning from P.**

Each procedure invocation causes creation of an *activation record* **or** *frame* **to hold such runtime information.**

State
Local x
Local y
⋮
Dynamic Link
Static Link
Parameters
Return Value

It's convenient to have each local occupy a fixed amount of storage in the frame. Therefore, arrays and other large objects are often indirectly accessed from a procedure's frame, with the actual storage allocated on stack after the frame.

A simple runtime storage layout



Since there are two dynamically growing areas, a simple scheme is to place these at opposite ends of the address space.

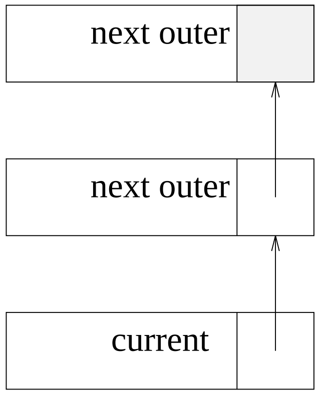
Following the contour of procedure entry and exit, activation records are usually allocated on a stack. Where languages allow suspension and resumption of procedures (*e.g.*, via *continuations*), then frames are garbage-collected from the heap when dead.

The stack can also be used for performing intermediate computations.

The heap is generally managed by some form of explicit or implicit garbage collection (4).

Access to nonlocals

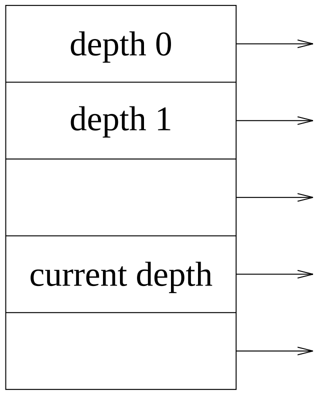
Static Links



At procedure entry, a link is inserted in the frame to a procedure's next outer scope, whose frame is linked to its outer scope, and so on. The static link is deallocated along with the frame.

Establishing the link is fast, but accessing the kth enclosing scope requires k indirections using static links. However, a good register allocator would cache these in the procedure's registers.

Displays



A display is an array of frame pointers. At procedure entry, the display is adjusted so that the frame at static depth d is accessed via entry d of the display. The display must be reset at procedure return.

Maintaining the display takes more time than with static links, but access to outer scopes is faster once the display is established.

Most programs make almost exclusive use of local and outermost scopes, with scant use of intermediate scopes. This is especially true in C, where the language offers no access to intermediate scopes except by explicit pointers.

Code Generation

As with parsing, methods for code generation can be classified:

Ad hoc.

As with semantic processing, code could be generated after the parse by traversing the AST. Typically, a combined pre- and post-order traversal suffices, where a node's type prompts the code generator to emit a parameterized template of code.

Systematic

- **Grammar-based (21).**
- **Grammars with attributes (20).**
- **Tree pattern-matching (17, 19).**
- **By peephole processing (11).**

These methods spend more time on instruction selection than can be afforded or managed by *ad hoc.* methods.

In a compiler course, the choice of code generation strategy is key to a successful experience. Many courses stop just before code generation, in which case the students do not experience the elation of watching their compilers actually work.

If the target of translation is reasonably high-level (*e.g.*, a LISP-like intermediate langauge), then *ad hoc.* methods are feasible. In this case, an interpreter should be provided to execute the translated programs.

Otherwise, experience with an automatic code generator is more beneficial. Watch for developments in the `lcc` system (18), which can be obtained by contacting Dave Hanson (`drh@princeton.edu`). If the MIPS instruction set were targeted, then Larus's SPIM simulator (30)(Appendix A) can greatly facilitate debugging the generated code.

Example of a high-level intermediate language

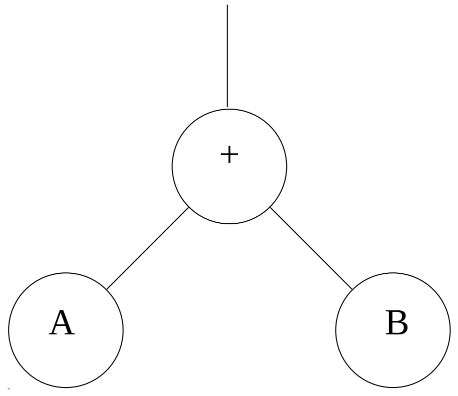
The language FRIL (10) was developed to ease code generation, primarily by resembling LISP and by offering a declarative mechanism for storage association. Each symbol FRIL is declared at most once as any procedure's local or parameter. Each "expression" declares the static depth of its frame, and provides a pointer to its outer scope.

```
int a1;
extern int a2;
int one;
void main() {
   int i;

   int factorial(X)
   int X;
   {
     int Y;
     Y = X;
     if (Y > 0) Y*factorial(X-1);
     else one;
   }

   one = 1;
   a1 = factorial(i=5);
   a2 = factorial(3);
```

```
(Expression 1  /* factorial */
  (PushLevel 5 (LinkExpressionID 2)
    (Args   (SymbolID 7)   /* X */)
    (Locals (SymbolID 8)   /* Y */)
  )
  (Def (SymbolID 8)   /* Y */
    (Use (SymbolID 7)   /* X */)
  )
  (-> 0
    (CHOOSE
      (
        (NE 0
           (GT (Use (SymbolID 8) /* Y */) 0)
        )

          (TIMES
            (Use (SymbolID 8) /* Y */)
            (-> 1
              (MINUS
                (Use (SymbolID 7) /* X */)
                1
              )
            )
          )
      )
      (1 (Use (SymbolID "one")))
    )
  )
)
```

Ad hoc. methods

For example, for a binary $+$ node, the code generator would be called recursively to place the result of the left and right subtrees in two known locations (say, registers R_1 and R_2). Code would then be emitted to form the sum, placing the result in yet another known location.



```
(PLUS
   /* Code for A */
   /* Code for B */
)
```

I usually provide procedures for generating FRIL's symbol table, for generating a PushLevel, and for indenting and formatting the output. The students must decide what constitutes an expression. For example, FRIL has only one control transfer operator: the procedure call. Thus, the body of an iterative loop must be invoked recursively to achieve iteration.

Students write some 200 lines of code to complete the *ad hoc.* code generator for FRIL.

A systematic method — Tree Rewriting

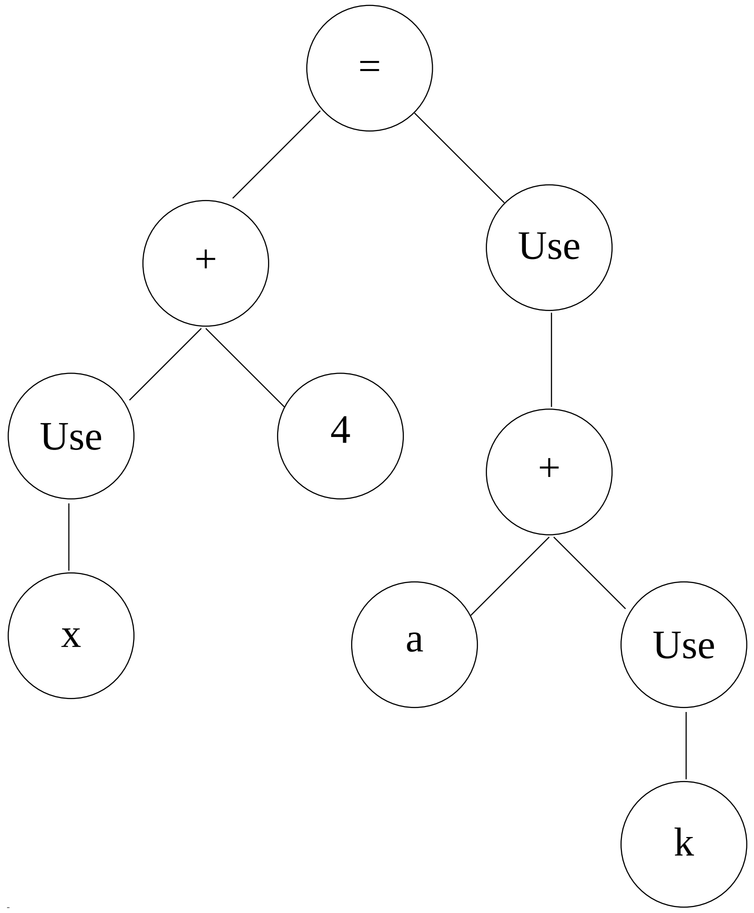
While the *ad hoc.* method services the AST a node at a time, tree rewriting systems can examine larger subtrees and searching for more optimal instruction sequences.

The AST shown to the right is representative of the code fragment

```
∗(x+4)=a[k];
```

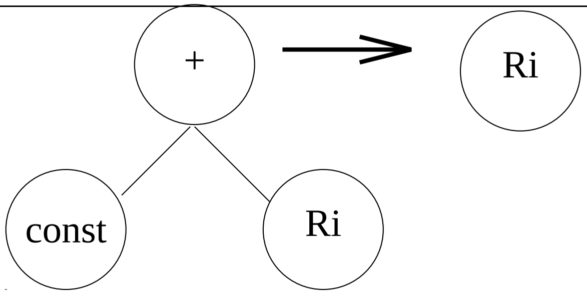
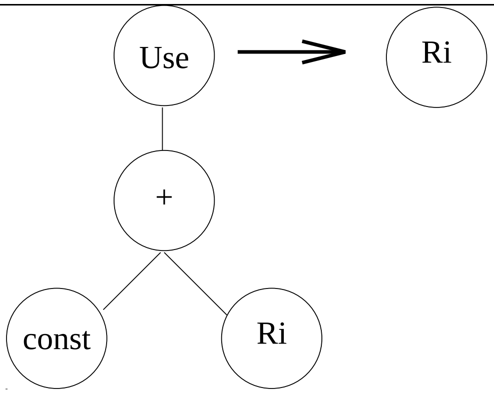
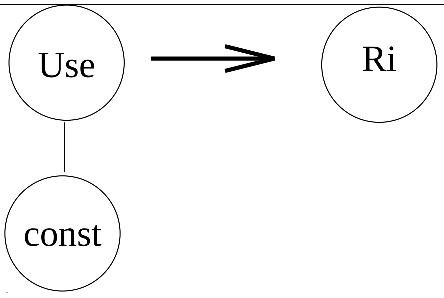
Note that left and right value analysis has already taken place.

Let's assume that from the perspective of code generation, the nodes x**,** a**, and** k **represent constants. This would be the case had the compiler assigned storage to these variables. If not, then the AST should reflect a level of indirection (probably off a popular register) to reach those variables.**



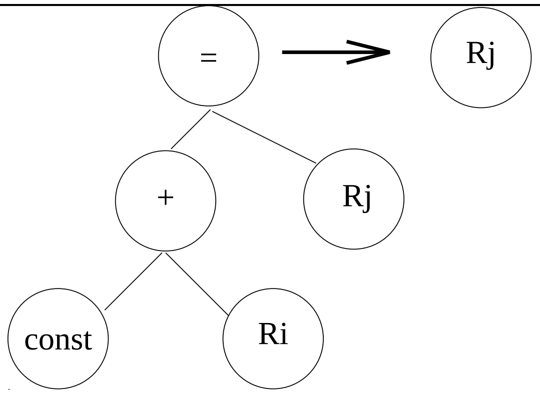
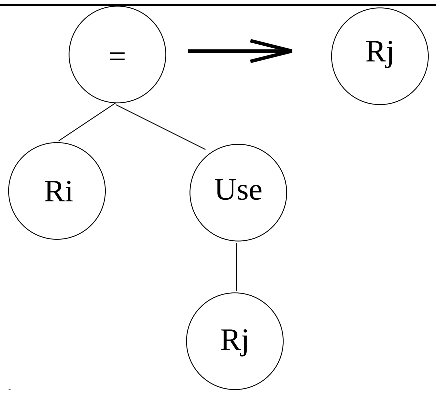
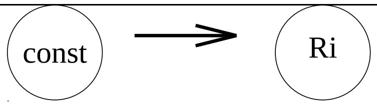
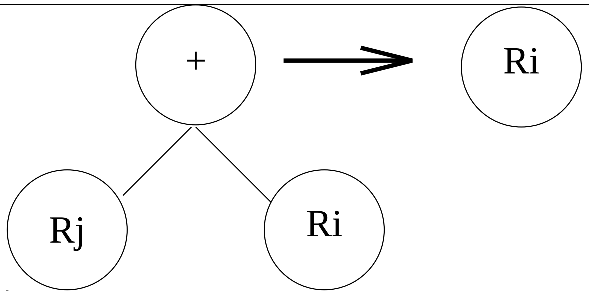
Given the richness of most instruction sets, trying all combinations of instructions to cover the tree would be prohibitively expensive. Most tree matching algorithms use *dynamic programming*, so that results previously holding for some subtree can be reused without additional cost.

Tree rewriting

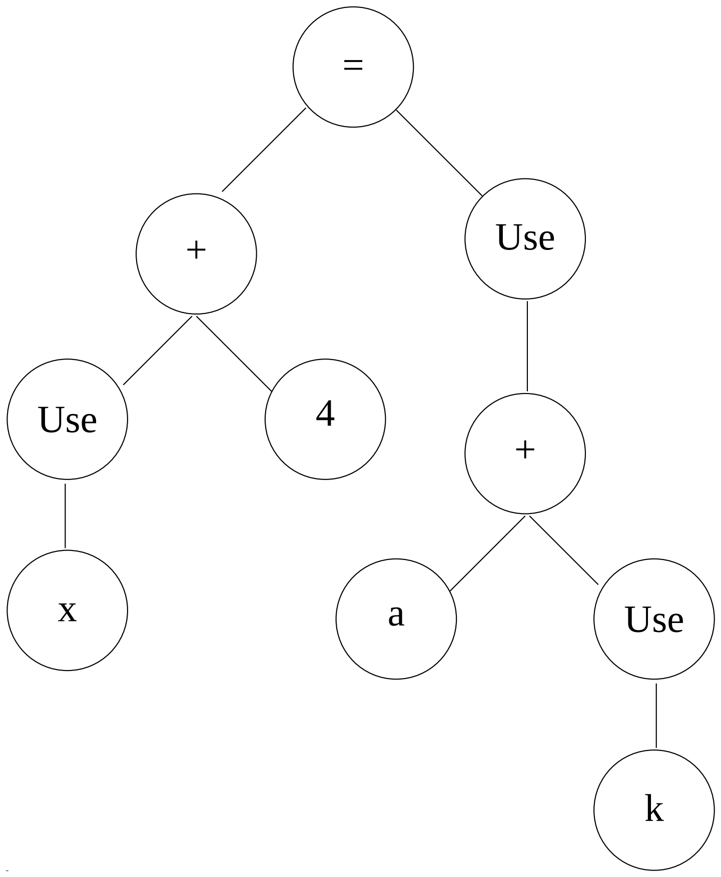
Rule	Rewrite	Instruction	Cost
1		$R_i \leftarrow R_i$**+const**	1
2		$R_i \leftarrow$ **M(**R_i**+const)**	5
3		$R_i \leftarrow$ **M(const)**	3

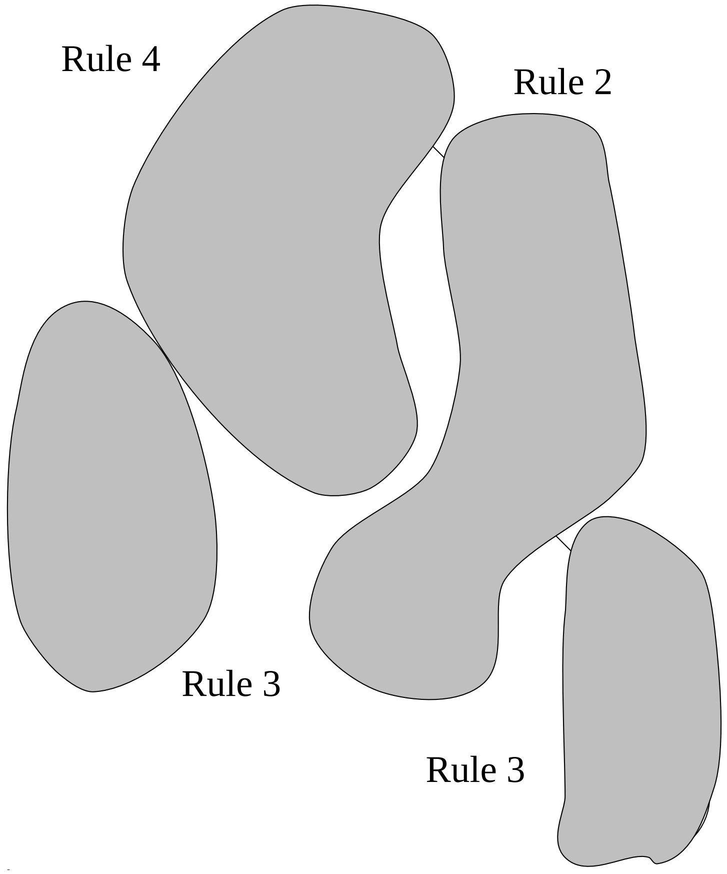
Not shown are the rules that account for the symmetry of addition.

Tree rewriting

Rule	Rewrite	Instruction	Cost
4		M(R_i+const) $\leftarrow R_j$	5
5		M(R_i) $\leftarrow$ M(R_j)	6
6		$R_i \leftarrow$ const	1
7		$R_i \leftarrow R_i$+R_j	1

Example – one way to cover the nodes

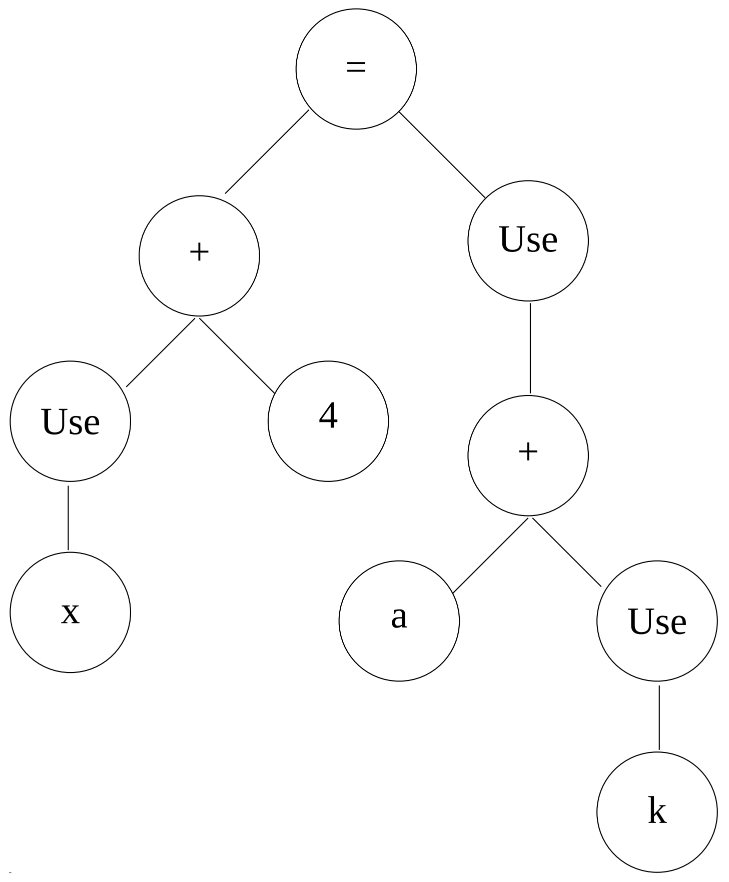


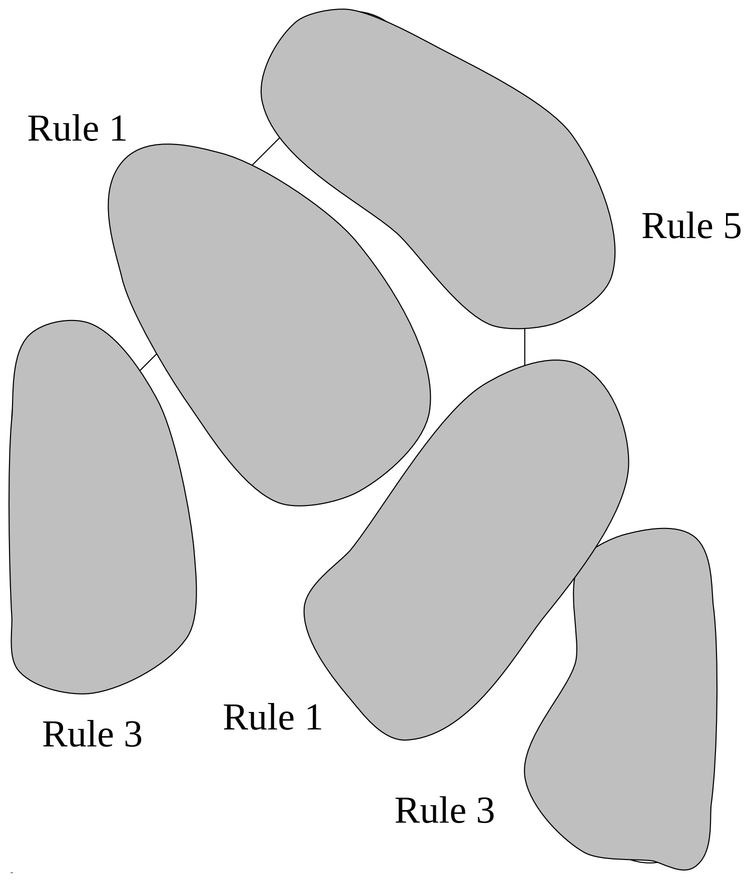


Rule	Instr	Cost
2	$R_i \rightarrow$M(R_i+const)	5
3	$R_i \rightarrow$M(const)	3
4	M(R_i+const)$\rightarrow R_j$	5

Rule	Cost
3	3
2	5
3	3
4	5
	16

Example – another way to cover the nodes

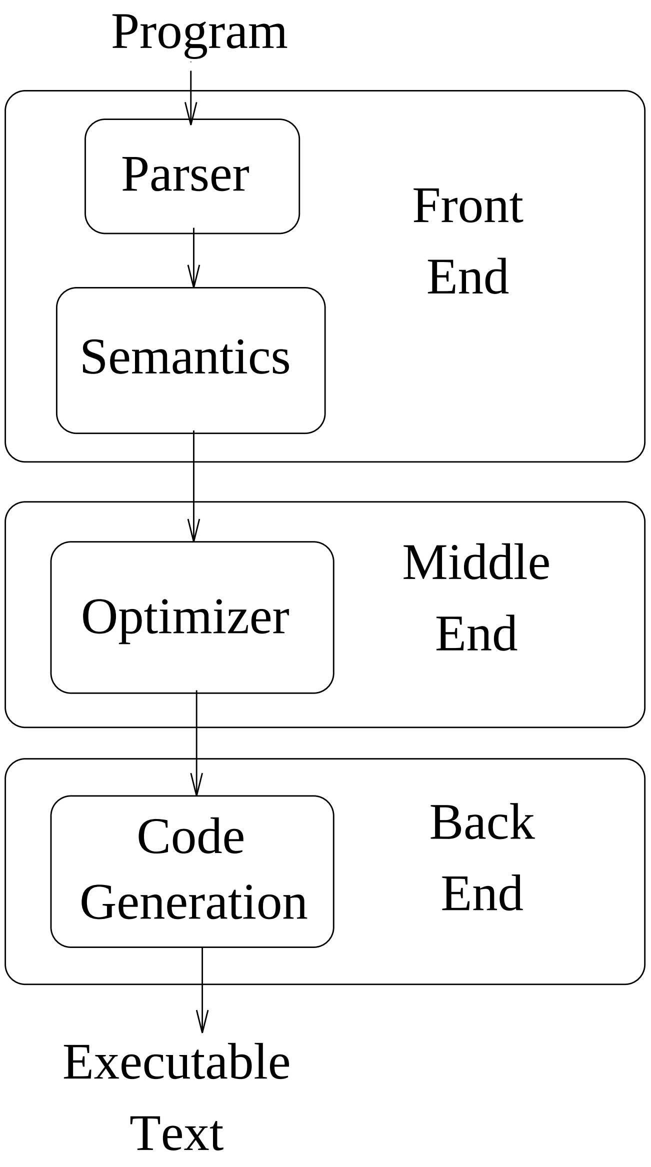




Rule	Instr	Cost
1	$R_i \rightarrow R_i$+const	1
3	$R_i \rightarrow$M(const)	3
5	M(R_i)$\rightarrow$M(R_j)	6

Rule	Cost
3	3
1	1
3	3
1	1
5	6
	14

Compiler organizations



Front end: Operator and storage abstractions, alias mechanisms.

Middle end:

- Dead code elimination
- Code motion
- Reduction in strength
- Constant propagation
- Common subexpression elimination
- Fission
- Fusion
- Strip mining
- Jamming
- Splitting
- Collapsing

Back end: Finite resource issues and code generation.

Some thoughts

Misconceptions

Optimization optimizes your program. There's probably a better algorithm or sequence of program transformations. While optimization hopefully improves your program, the result is usually not optimal.

Optimization requires (much) more compilation time. For example, dead code elimination can reduce the size of program text such that overall compile time is also reduced.

A clever programmer is a good substitute for an optimizing compiler. While efficient coding of an algorithm is essential, programs should not be obfuscated by "tricks" that are architecture- (and sometimes compiler-) specific.

All too often...

Optimization is disabled by default. Debugging optimized code can be treacherous (45, 23). Optimization is often the primary suspect of program misbehavior—sometimes deservedly so. "No, not the *third* switch!"

Optimization is slow. Transformations are often applied to too much of a program. Optimizations are often textbook recipes, applied without proper thought.

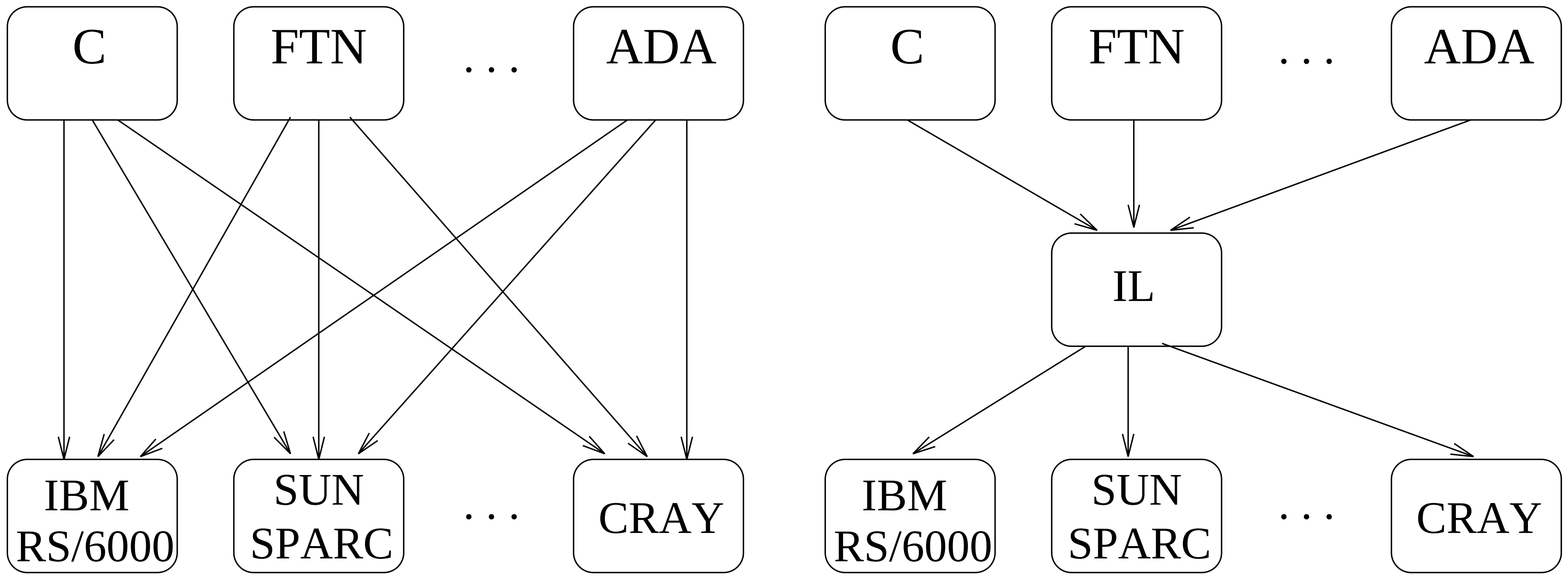
Optimization produces incorrect code. Although recent work is encouraging (42), optimizations are usually developed *ad hoc.*

Programmers are trained by their compilers. A style is inevitably developed that is conducive to optimization.

Optimization is like *sex:*

- Everybody claims to get good results using *exotic* techniques;
- Nobody is willing to provide the details.

Multilingual systems



Architecting an *intermediate language* reduces the incremental cost of accommodating new source languages or target architectures (5). Moreover, many optimizations can be performed directly on the intermediate language text, so that source- and machine-independent optimizations can be performed by a common middle-end.

Intermediate languages

It's very easy to devote much time and effort toward choosing the "right" IL. Below are some guidelines for choosing or developing a useful intermediate language:

- **The IL should be a** *bona fide* **language, and not just an aggregation of data structures.**
- **The semantics of the IL should be cleanly defined and readily apparent.**
- **The IL's representation should not be overly verbose:**
  - **Although some expansion is inevitable, the IL-to-source token ratio should be as low as possible.**
  - **It's desirable for the IL to have a verbose, human-readable form.**
- **The IL should be easily and cleanly extensible.**
- **The IL should be sufficiently general to represent the important aspects of multiple front-end languages.**
- **The IL should be sufficiently general to support efficient code generation for multiple back-end targets.**

A sampling of difficult issues:

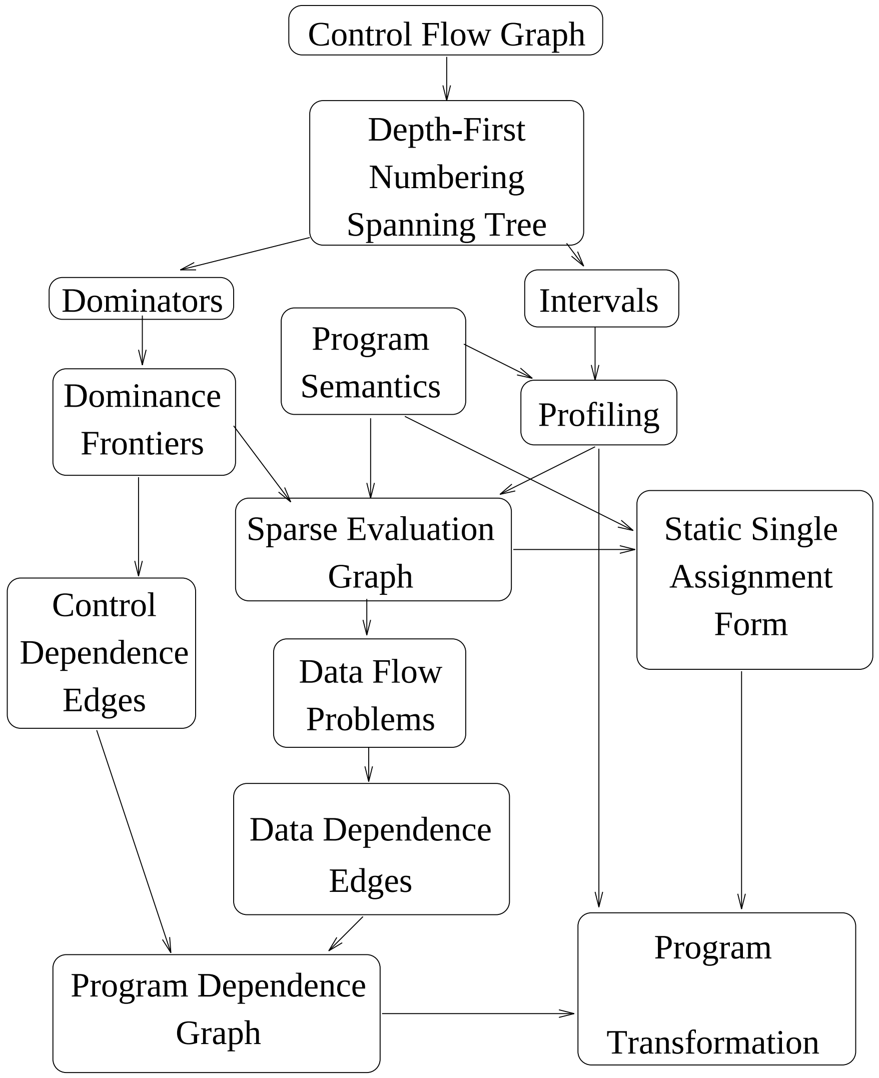
- **How should a string operation be represented (intact or as a "loop")?**
- **How much detail of a procedure's behavior is relevant?**

Ideally, an IL has *fractal* **characteristics: optimization can proceed at a given level; the IL can be "lowered"; optimization is then applied to the freshly exposed description.**

What happens in the middle end?

Essentially, the program is transformed into an observably equivalent while less resource-consumptive program. Such transformation is often based on:

- **Assertions provided by the program author or benefactor.**
- **The program dependence graph (29, 15, 6).**
- **Static single assignment (SSA) form (8, 3, 44, 9).**
- **Static information gathered by solving data flow problems (25, 34, 35, 36, 22, 37, 38, 27).**
- **Run-time information collected by** *profiling* **(40).**



Let's take a look at an example that benefits greatly from optimization. . .

Unoptimized matrix multiply

```
for $i = 1$ to $N$ do
    for $j = 1$ to $N$ do
        $A[i,j] \leftarrow 0$
        for $k = 1$ to $N$ do
            $A[i,j] \leftarrow A[i,j] + B[i,k] \times C[k,j]$
        od
    od
od
```

Note that $A[i,j]$ is really

$$Addr(A) + ((i-1) \times K_1 + (j-1)) \times K_2$$

which takes 6 integer operations.

The innermost loop of this "textbook" program takes

24	integer ops
3	loads
1	floating add
1	floating mpy
1	store
30	instructions

Optimizing matrix multiply

```
for $i = 1$ to $N$ do
    for $j = 1$ to $N$ do
        $a \leftarrow \&(A[i,j])$
        for $k = 1$ to $N$ do
            $\star a \leftarrow \star a + B[i,k] \times C[k,j]$
        od
    od
od
```

The expression $A[i,j]$ is *loop-invariant* with respect to the k loop. Thus, *code motion* can move the address arithmetic for $A[i,j]$ out of the innermost loop.

The resulting innermost loop contains only 12 integer operations.

```
for $i = 1$ to $N$ do
    $b \leftarrow \&(B[i,1])$
    for $j = 1$ to $N$ do
        $a \leftarrow \&(A[i,j])$
        for $k = 1$ to $N$ do
            $\star a \leftarrow \star a + \star b \times C[k,j]$
            $b \leftarrow b + K_B$
        od
    od
od
```

As loop k iterates, addressing arithmetic for B changes from $B[i,k]$ to $B[i,k+1]$. *Induction variable analysis* detects the constant difference between these expressions.

The resulting innermost loop contains only 7 integer operations.

Similar analysis for C yields only 2 integer operations in the innermost loop, for a speedup of nearly 5. We can do better, especially for large arrays.

If optimization is. . .

so great because:

A good compiler can sell (even a slow) machine. Optimizing compilers easily provide a factor of two in performance. Moreover, the analysis performed during program optimization can be incorporated into the "programming environment" (29, 7, 43).

New languages and architectures motivate new program optimizations. Although some optimizations are almost universally beneficial, the advent of functional and parallel programming languages has increased the intensity of research into program analysis and transformation.

Programs can be written with attention to clarity, rather than performance. There is no substitute for a good algorithm. However, the expression of an algorithm should be as independent as possible of any specific architecture.

then:

Why does it take so long? Compilation time is usually 2–5 times slower, and programs with large procedures often take longer. Often this is the result of poor engineering: better data structures or algorithms can help in the optimizer.

Why does the resulting program sometimes exhibit unexpected behavior? Sometimes the source program is at fault, and a bug is uncovered when the optimized code is executed; sometimes the optimizing compiler is itself to blame.

Why is "no-opt" the default? Most compilations occur during the software development cycle. Unfortunately, most debuggers cannot provide useful information when the program has been optimized (45, 23). Even more unfortunately, optimizing compilers sometimes produce incorrect code. Often, insufficient time is spent testing the optimizer, and with no-opt the default, bugs in the optimizer may remain hidden.

Ingredients in a data flow framework

Data flow graph

$$\mathcal{G}_{df} = (\mathcal{N}_{df}, \mathcal{E}_{df})$$

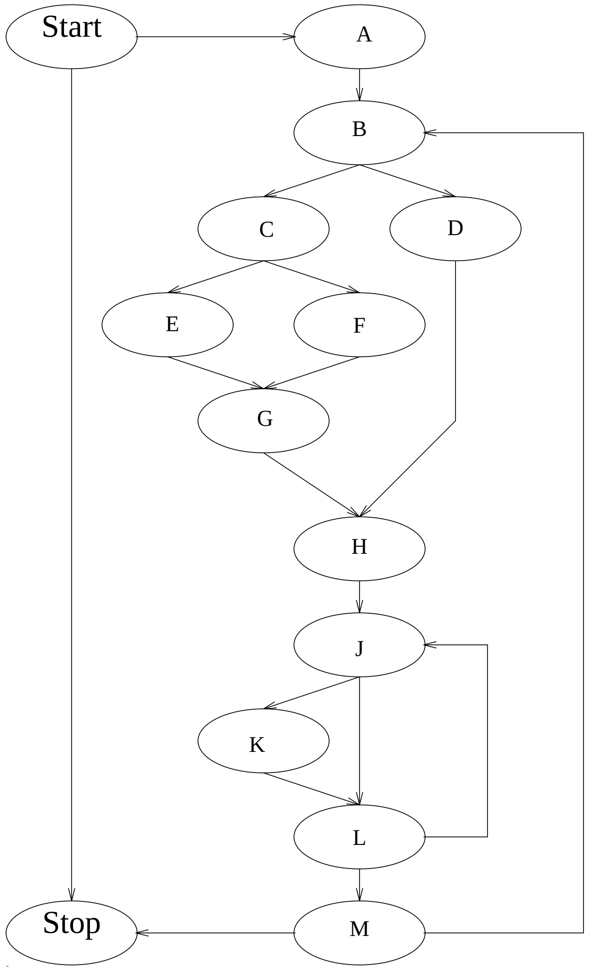
which is based on a directed *flow graph* $\mathcal{G}_f = (\mathcal{N}_f, \mathcal{E}_f)$, **typically the** *control flow graph* **of a procedure.**

A data flow problem is

forward if the solution at a node may depend only on the program's past behavior;

backward if the solution at a node may depend only on a program's future behavior;

bidirectional if both past and future behavior is relevant (12, 13, 14).



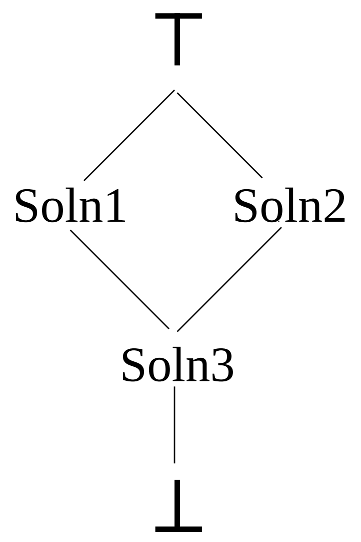
- We'll assume the data flow graph is augmented with a $Start$ and $Stop$ node, and an edge from $Start$ to $Stop$.
- We'll limit our discussion to non-bidirectional problems, and assume that edges in the data flow graph are oriented in the direction of the data flow problem.

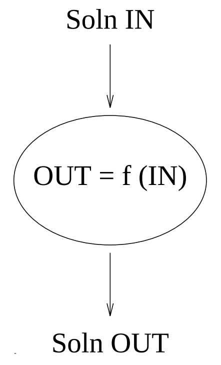
Ingredients in a data flow framework (cont'd)

Meet lattice which determines the outcome when disparate solutions combine. The lattice is specified with distinguished elements

- $\top$ **which represents the best possible solution, and**
- $\perp$ **which represents the worst possible solution.**

Transfer Functions which transform one solution into another.





We'll use the meet lattice to summarize the effects of convergent paths in the data flow graph, and transfer functions to model the effects of a data flow graph path on the data flow solution.

We'll begin with some simple *bit-vectoring* data flow problems, classically solved as operations on bit-vectors. For ease of exposition, we'll associate data flow solutions with the edges, rather than the nodes, of the data flow graph.

Available expressions

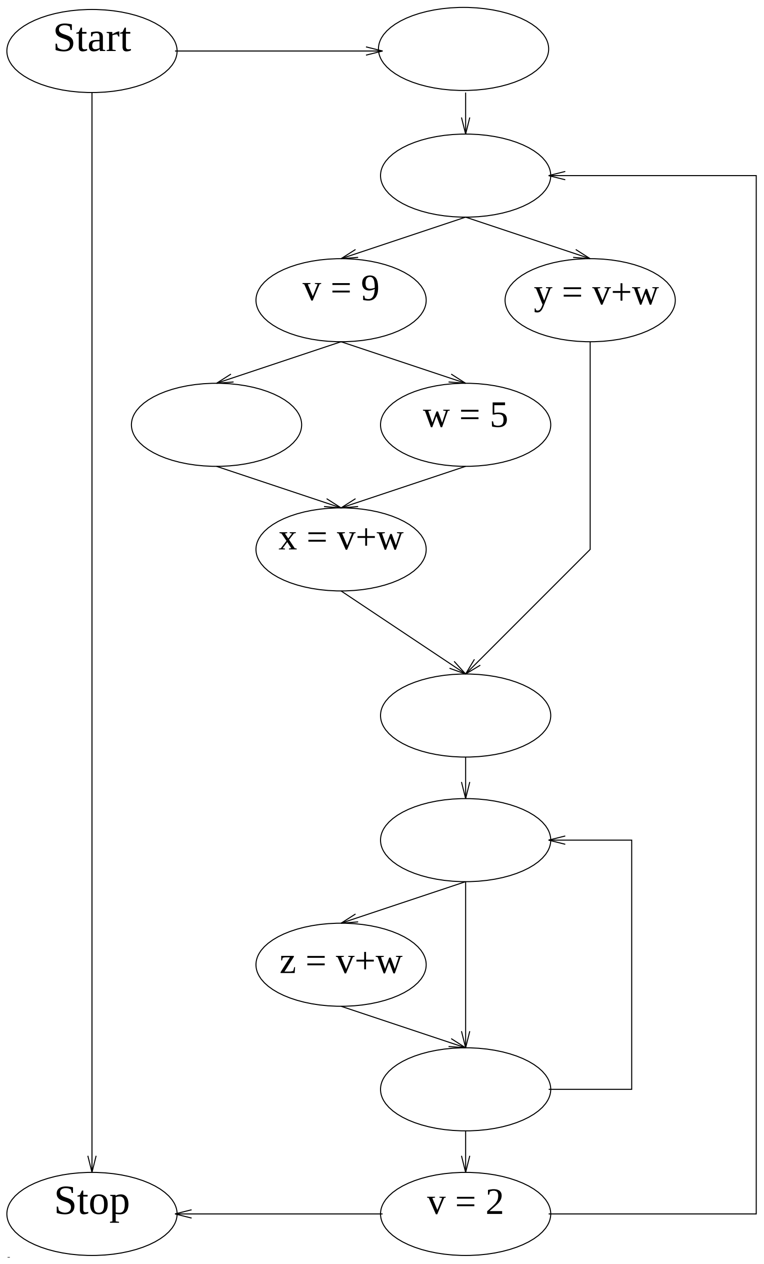
An expression $expr$ is *available* ($Avail$) at flow graph edge e if any past behavior of the program includes a computation of the value of $expr$ at e.

Consider the expression $(v + w)$ in the flow graph shown to the right. If the expression is available at the assignment to z, then it need not be recomputed.

- This is a forward problem, so the data flow graph will have the same edges and $Start$ and $Stop$ nodes as the flow graph.
- The solution for any given $expr$ is either $Avail$ or $\overline{Avail}$.
- The "best" solution for an expression is $Avail$. We thus obtain the two-level lattice:

  $\top$ is $Avail$.

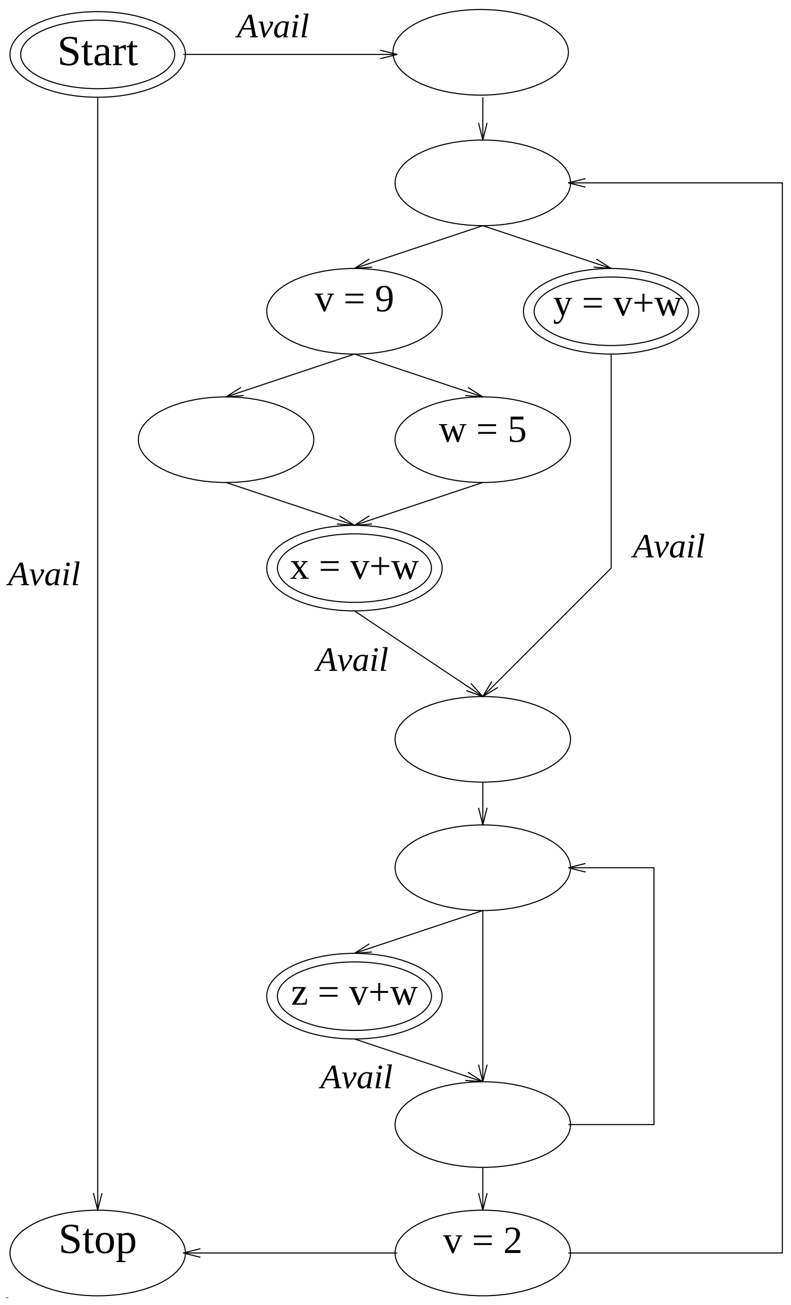
  $\perp$ is $\overline{Avail}$.



Available expressions(cont'd)

Nodes that compute an expression make that expression available. We also assume that every expression is available from $Start$.

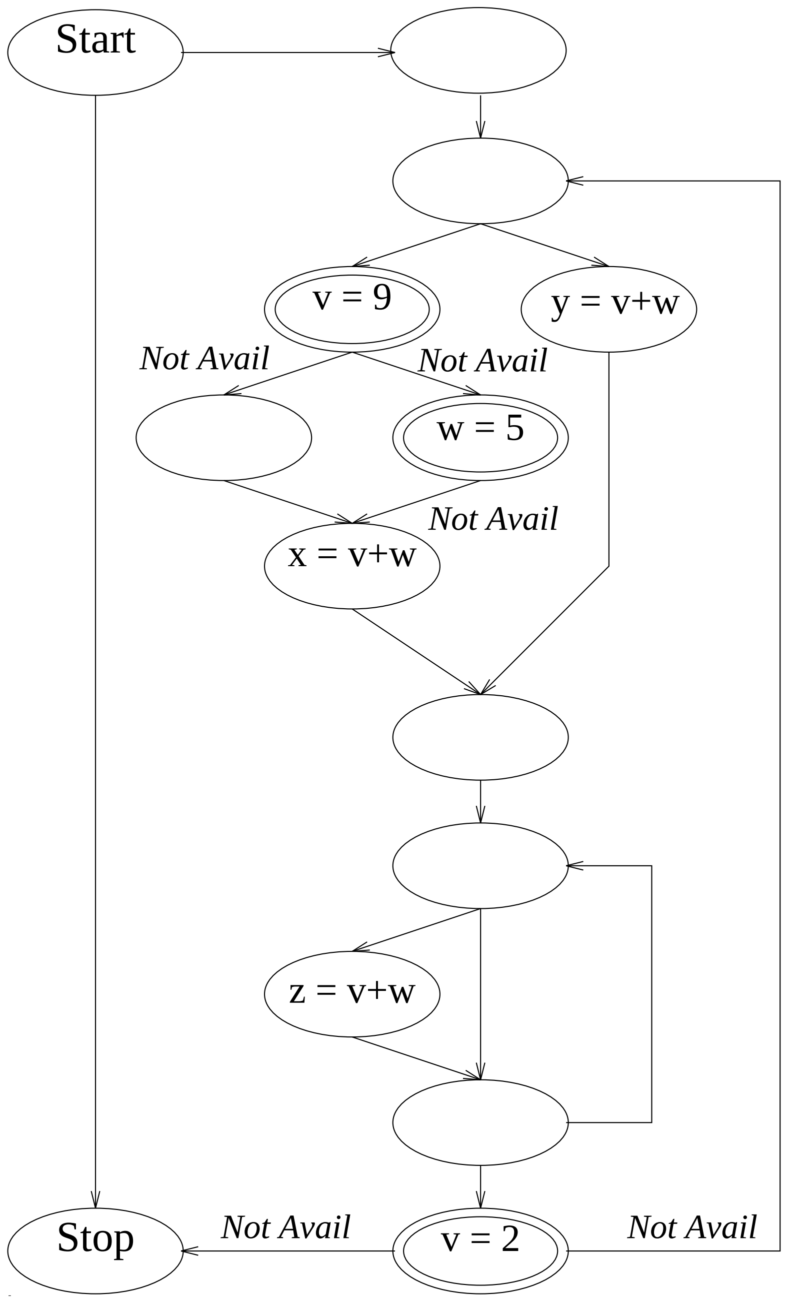
The transfer function for each highlighted node makes the expression $(v + w)$ $Avail$, regardless of the solution present at the node's input.



Available expressions(cont'd)

Nodes that assign to any variable in an expression make that expression not available, even if the variable's value is unchanged.

The transfer function for each highlighted node makes the expression $(v + w)$ $\overline{Avail}$, regardless of the solution present at the node's input.



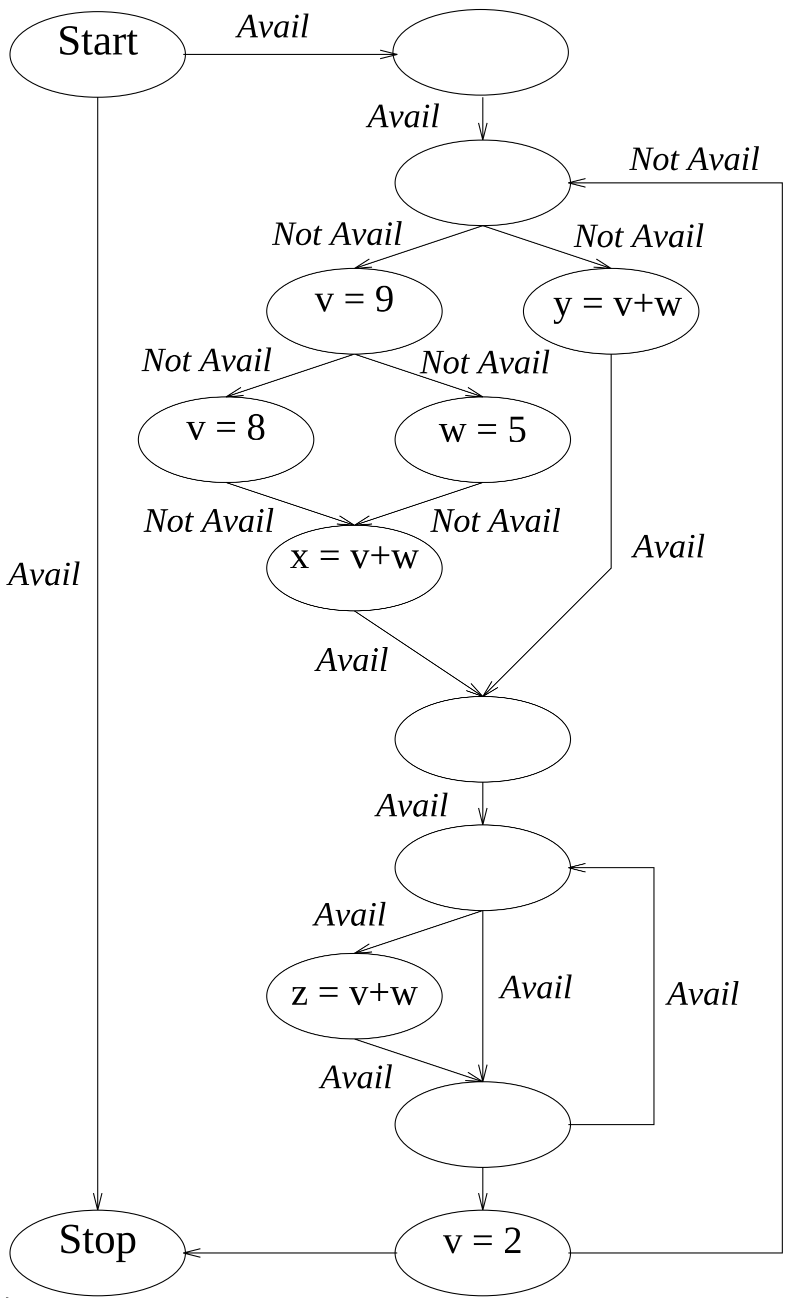
Available expressions (cont'd)

Here we see the global solution for availability of the expression $(v + w)$.

Each of the highlighted nodes shown previously asserts a solution on its output edge(s). It's the job of global data flow analysis to assign the best possible solution to every edge in the data flow graph, consistent with the asserted solutions.

The expression $(v + w)$ need not be computed in the assignment to z. The relevant value is held either in x, or y, depending on program flow.

To solve this problem using bit-vectors, assign each expression a position in the bit-vector. When an expression is available, its associated bit is 1.



Live variables

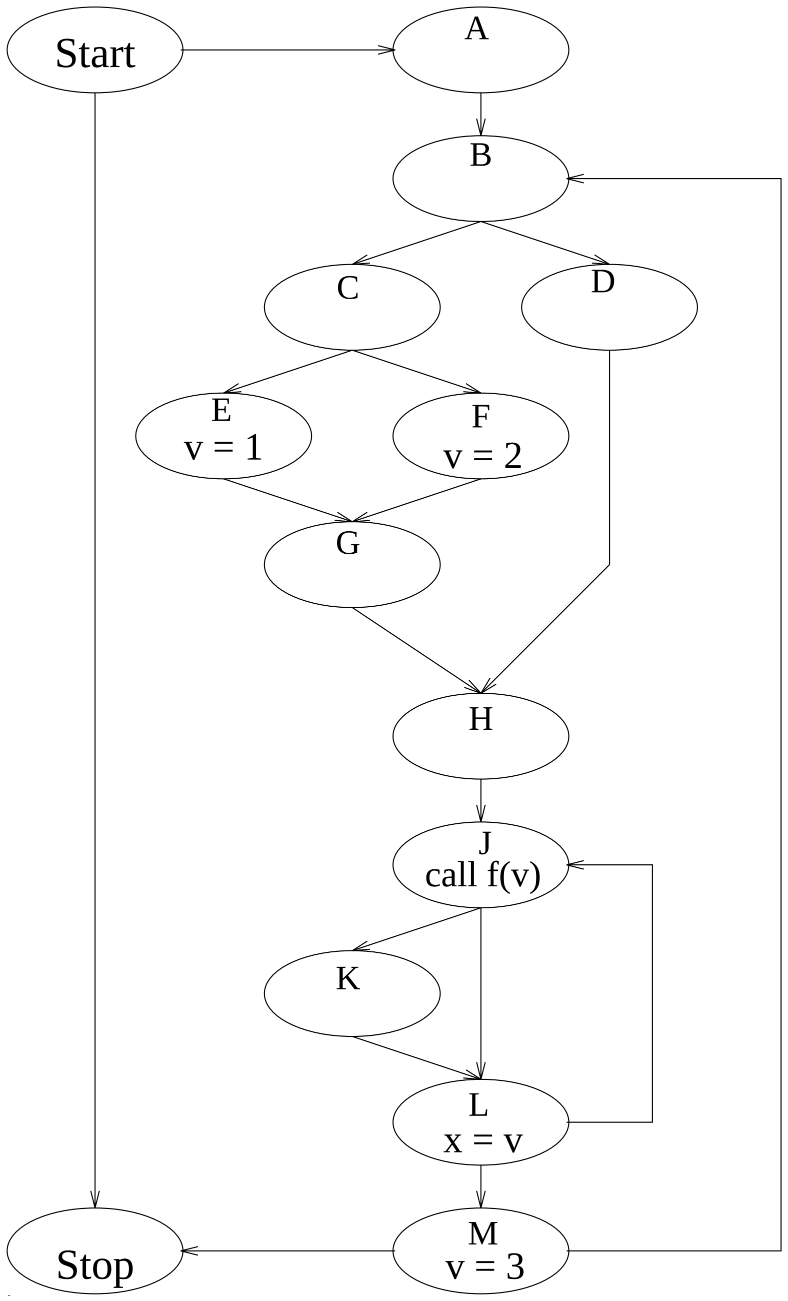
A variable v is *live* at edge e if the future behavior of the program may reference the value of v at e.

If a variable v is not live, then any resources associated with v (registers, storage, etc.) may be reclaimed.

- **This is a backward problem.**
- **In the bit-vector representation, each variable is associated with a bit.**
- **The "best" solution is $\overline{Live}$, so we obtain the two-level lattice:**

  $\top$ is $\overline{Live}$.

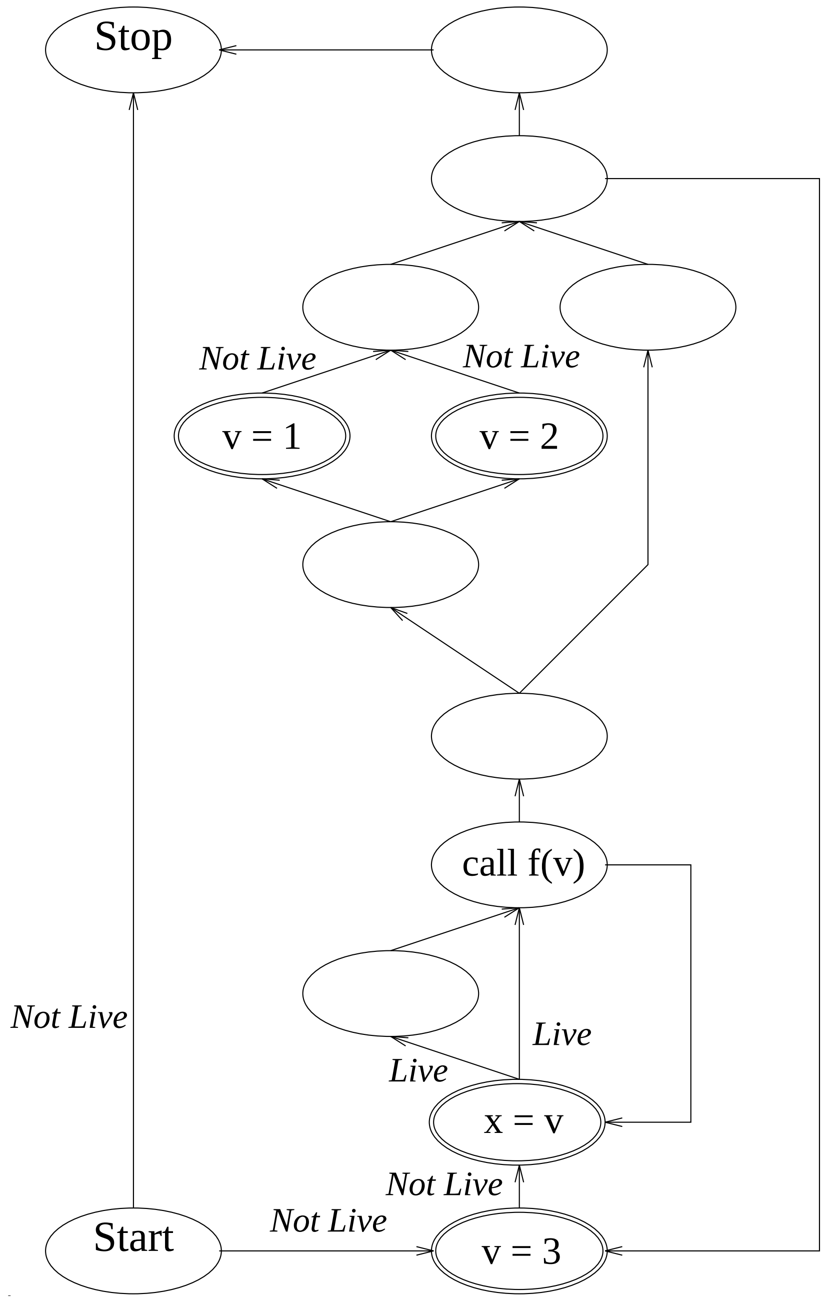
  $\bot$ is $Live$.



Live variables (cont'd)

Each of the highlighted nodes affects the data flow solution:

- **If a node uses v, then the node's asserts that v is $Live$.**
- **If a node kills v, then the node's output asserts that v is $\overline{Live}$.**



Live variables (cont'd)

If a node Y preserves v (as might a procedure call), then the node does not affect the solution.

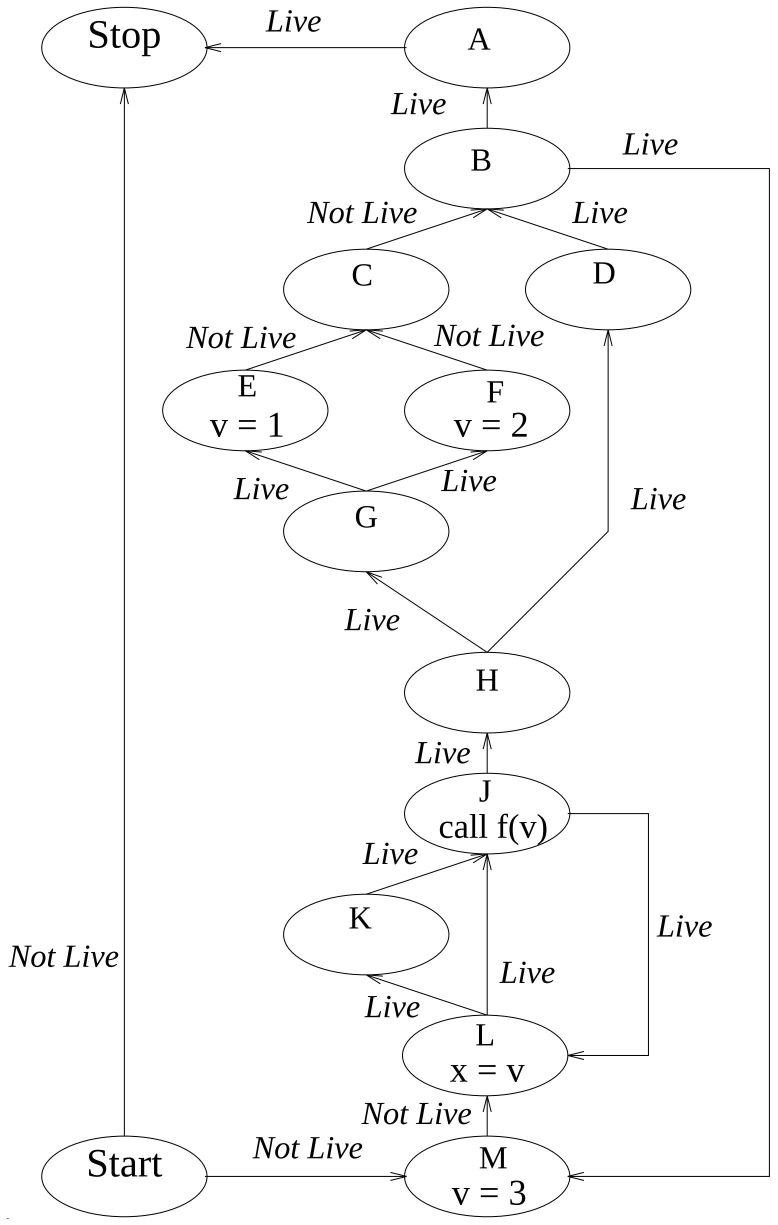
- **If v is $Live$ on "input" to Y, then Y cannot make v $\overline{Live}$.**
- **If v is $\overline{Live}$ on "input" to Y, then Y does not make v $Live$.**

Node Y's transfer function is therefore the *identity* function:

$$f_Y(IN) = IN$$

assuming node Y does not use v.

Global solution: Live variables



Formal specification of a data flow framework

The ***data flow graph***

$$\mathcal{G}_{df} = (\mathcal{N}_{df}, \mathcal{E}_{df})$$

has been described previously:

- **its edges are oriented in the direction of the data flow problem;**
- $\mathcal{G}_{df}$ **is augmented with nodes** $Start$ **and** $Stop$ **and an edge** $(Start, Stop)$**, suitably inserted with respect to the direction of the data flow problem.**

Successors and predecessors are also defined with respect to the direction of the data flow problem:

$$\begin{aligned} Succs(Y) &= \{ Z \mid (Y, Z) \in \mathcal{E}_{df} \} \\ Preds(Y) &= \{ X \mid (X, Y) \in \mathcal{E}_{df} \} \end{aligned}$$

The ***meet semilattice*** **is**

$$L = (A, \top, \bot, \preceq, \wedge)$$

A **is a set (usually a powerset), whose elements form the domain of the data flow problem,**

$\top$ and $\bot$ **are distinguished elements of** A**, usually called "top" and "bottom", respectively,**

$\preceq$ **is a reflexive partial order, and**

$\wedge$ **is the associative and commutative** ***meet*** **operator, such that for any** $a, b \in A$**,**

$$\begin{aligned} a \preceq b &\iff a \wedge b = a \\ a \wedge a &= a \\ a \wedge b &\preceq a \\ a \wedge b &\preceq b \\ a \wedge \top &= a \\ a \wedge \bot &= \bot \end{aligned}$$

These rules allow formal reasoning about $\top$ **and** $\bot$ **in a framework.**

Formal specification (cont'd)

The set F of *transfer functions*

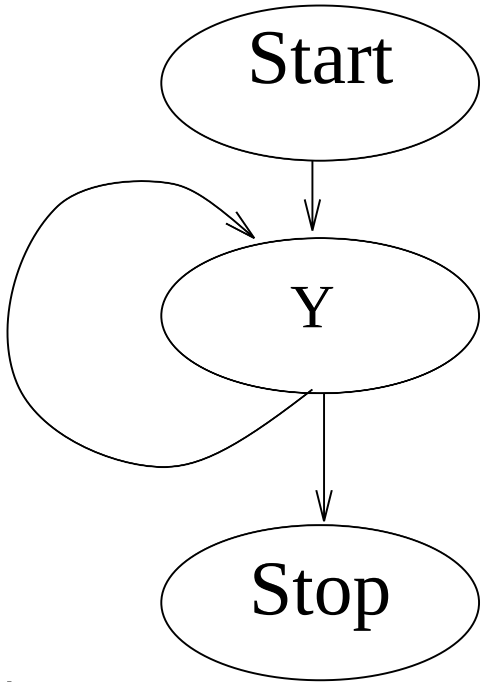
$$\mathcal{F} \subseteq \{f : L \mapsto L\}$$

has elements for describing the behavior of any flow graph node with respect to the data flow problem.

To obtain a stable solution, we'll require the functions in $\mathcal{F}$ to be *monotone***:**

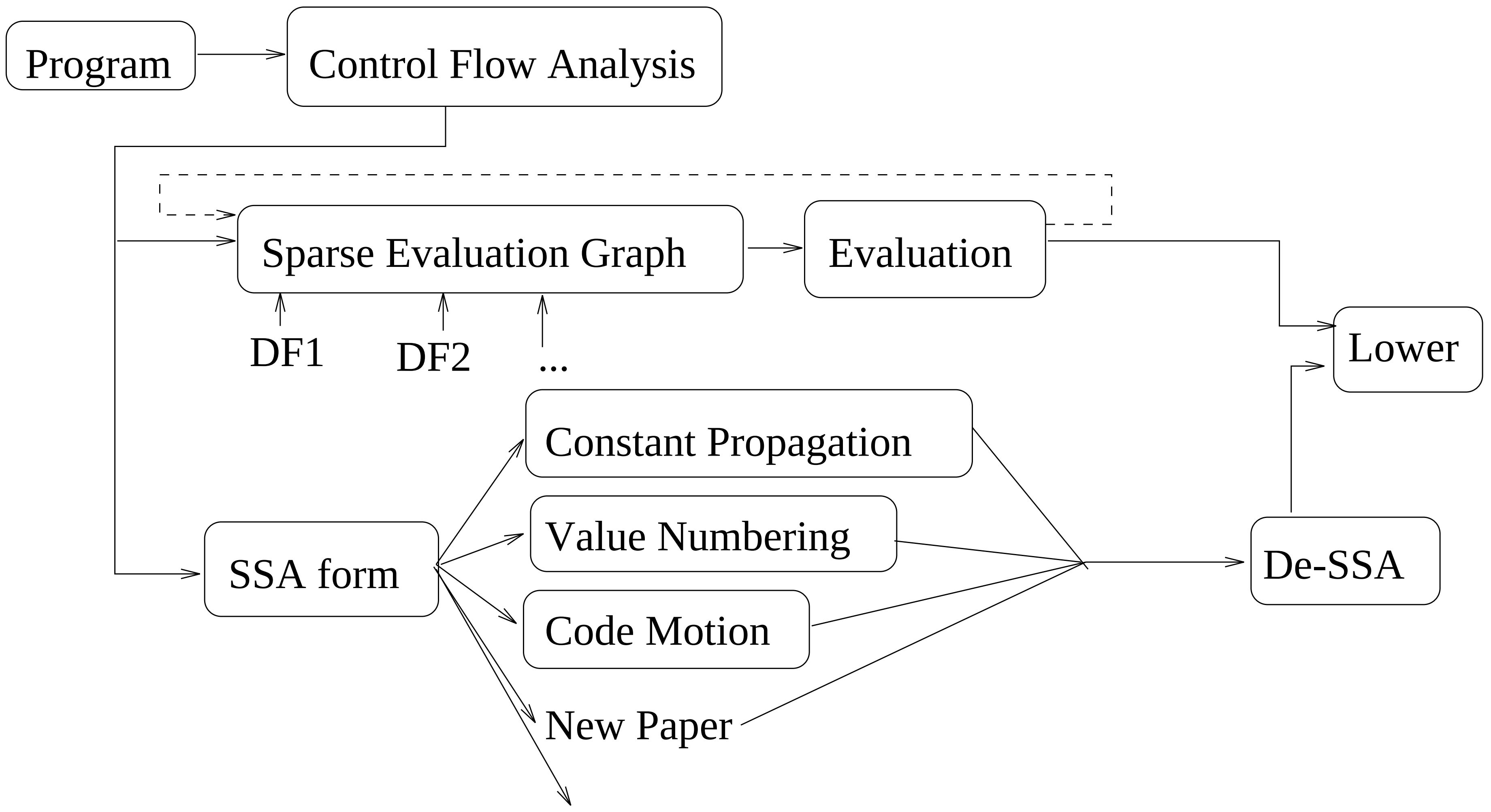
$$(\forall f \in \mathcal{F})(\forall x, y \in L)$$
$$x \preceq y \rightarrow f(x) \preceq f(y)$$

In other words, a node cannot produce a "better" solution when given "worse" input. Given a two-level lattice, evaluation of the data flow graph shown to the right oscillates between solutions and never reaches a fixed point.



$$f_Y(IN) = \begin{cases} \top & \textbf{if } IN = \bot \\ \bot & \textbf{if } IN = \top \end{cases}$$

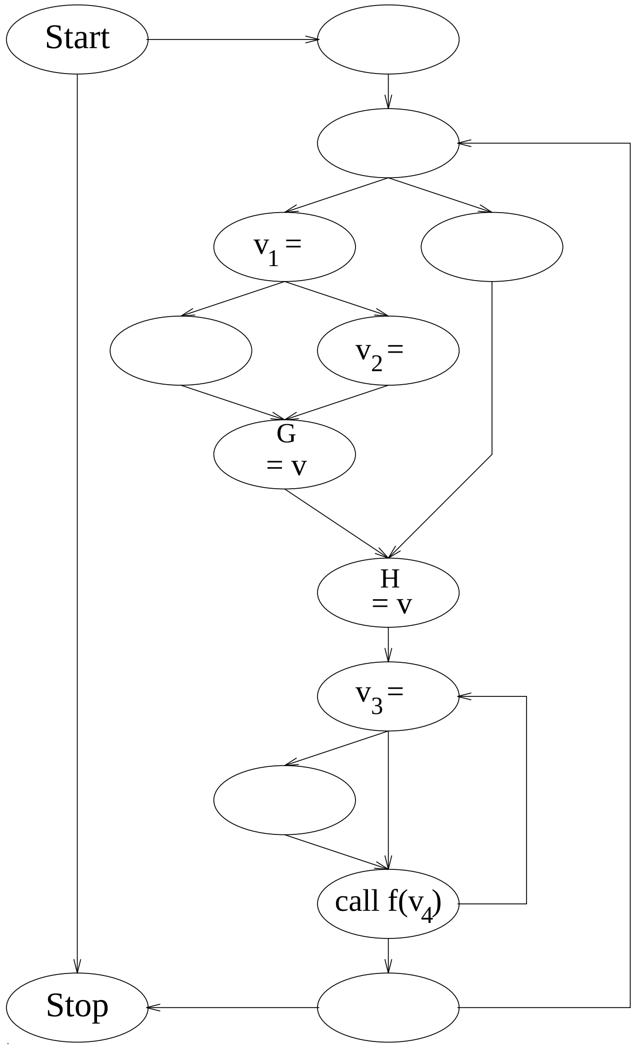
Big picture

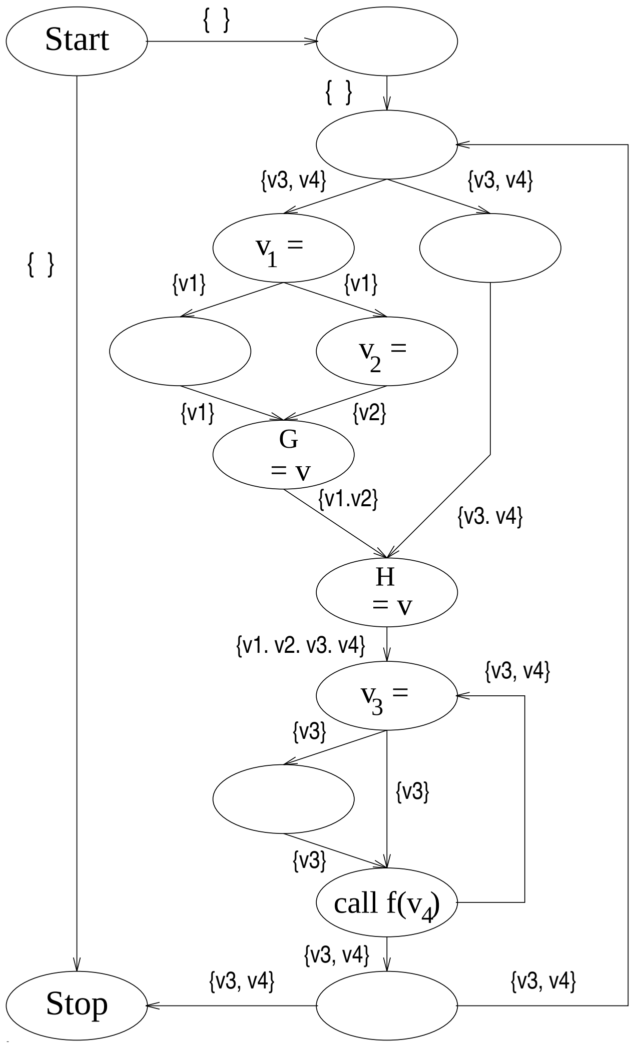


We'll now examine some special algorithms for optimization, based on a single assignment representation.

Static Single Assignment (SSA) form

Below are shown a program and its reaching definitions.



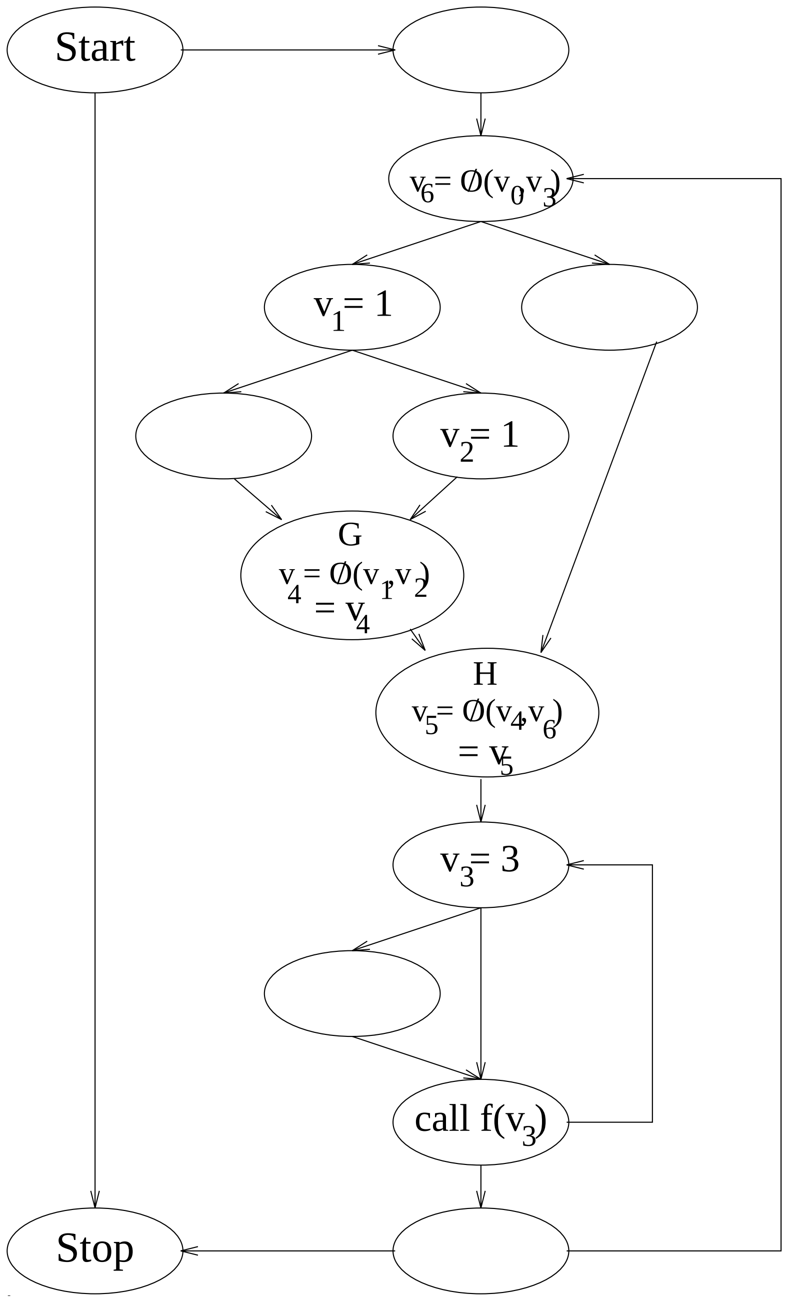


Notice how the use of v at G is reached by two definitions, and the use at H is reached by four definitions. If each use were reached by just a single definition, data flow analysis based on definitions could consult one definition per use.

SSA form (cont'd)

Here we see the SSA form of the program.

- **Each definition of v is with respect to a distinct symbol: v_1 is as different from v_2 as x would be from y.**
- **Where multiple definitions reach a node, a ϕ-function is inserted, with arguments sufficient to receive a different "name" for v on each in-edge.**
- **Each use is appropriately renamed to the distinct definition that reaches it.**
- **Although ϕ-functions could have been placed at every node, the program shown has exactly the right number and placement of ϕ-functions to combine multiple defs from the original program.**
- **Our example assumes that procedure f does not modify v.**


Start
$v_6 = \emptyset(v_0, v_3)$
$v_1 = 1$
$v_2 = 1$
G
$v_4 = \emptyset(v_1, v_2)$
$= v_4$
H
$v_5 = \emptyset(v_4, v_6)$
$= v_5$
$v_3 = 3$
call $f(v_3)$
Stop

SSA form (cont'd)

Each def is now regarded as a "killing" def, even those usually regarded as preserving defs. For example, if v is *potentially* modified by the call site, then the old value for v must be passed into the called procedure, so that its value can be assigned to the name for v that *always* emerges from the procedure.

```
Procedure $foo(v)$
    if $(c)$ then
        $v \leftarrow 7$
    else
        /* Do nothing */
    fi
end
```

```
Procedure $foo(v_{out}, v_{in})$
    $v_0 \leftarrow v_{in}$
    if $(c)$ then
        $v_1 \leftarrow 7$
    else
        /* Do nothing */
    fi
    $v_2 \leftarrow \phi(v_0, v_1)$
    $v_{out} \leftarrow v_2$
end
```

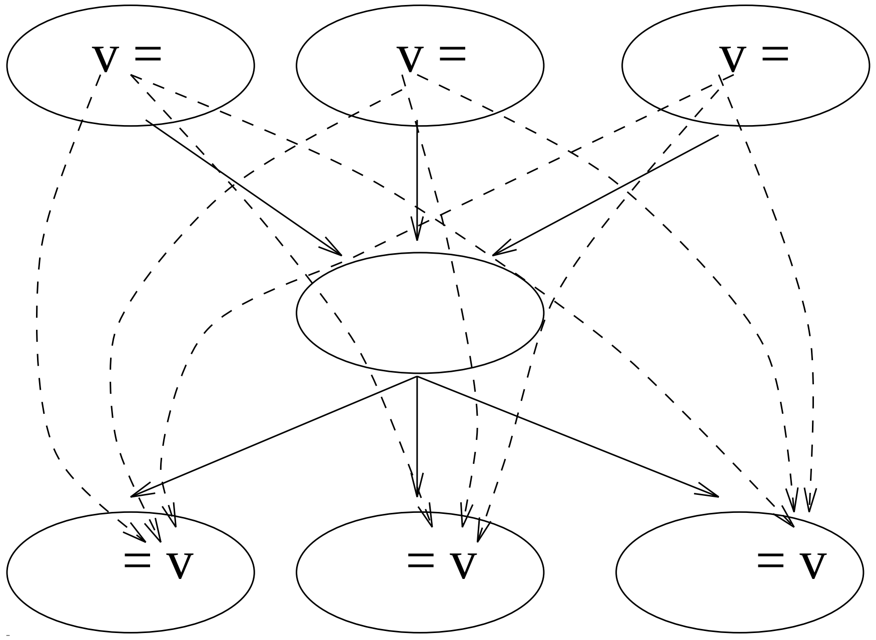
SSA form can be computed by a data flow framework, in which the transfer function for a node with multiple reaching defs of v generates its own def of v. Uses are then named by the solution in effect at the associated node.

SSA form construction [9]

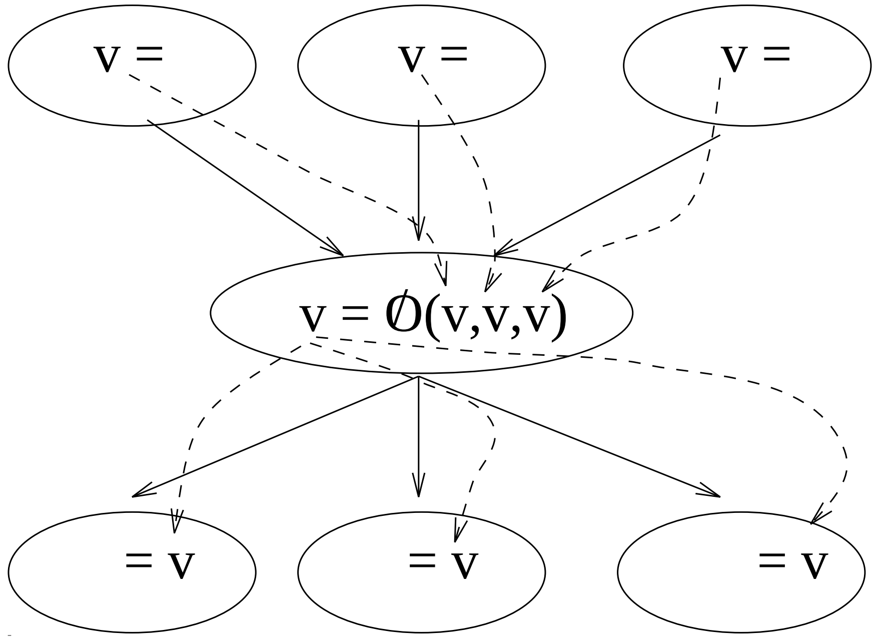
1. Every preserving def is turned into a killing def, by copying potentially unmodified values (at subscripted defs, call sites, aliased defs, etc.).
2. Each ordinary definition of v defines a new name.
3. At each node in the flow graph where multiple definitions of v *meet*, a ϕ-function is introduced to represent yet another new name for v.
4. Uses are renamed by their dominating definition (where uses at a ϕ-function are regarded as belonging to the appropriate predecessor node of the ϕ-function).

Why is SSA good?

Data flow algorithms built on def-use chains gain asymptotic efficiency as shown below:



Quadratic def-use chains



Linear def-use chains

With each use reached by a unique def, program transformations such as code motion are simplified: motion of a use depends primarily on motion of its unique reaching def. Intuitively, the program has been transformed to represent directly the flow of values. We'll now look at some optimizations that are simplified by SSA form.

SSA constant propagator [44]

Original Program

```
$i \leftarrow 6$
$j \leftarrow 1$
$k \leftarrow 1$
**repeat**
    **if** $(i = 6)$ **then**
        $k \leftarrow 0$
    **else**
        $i \leftarrow i + 1$
    **fi**
    $i \leftarrow i + k$
    $j \leftarrow j + 1$
**until** $(i = j)$
```

SSA form

```
$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
**repeat**
    $i_2 \leftarrow \phi(i_1, i_5)$
    $j_2 \leftarrow \phi(j_1, j_3)$
    $k_2 \leftarrow \phi(k_1, k_4)$
    **if** $(i_2 = 6)$ **then**
        $k_3 \leftarrow 0$
    **else**
        $i_3 \leftarrow i_2 + 1$
    **fi**
    $i_4 \leftarrow \phi(i_2, i_3)$
    $k_4 \leftarrow \phi(k_3, k_2)$
    $i_5 \leftarrow i_4 + k_4$
    $j_3 \leftarrow j_2 + 1$
**until** $(i_5 = j_3)$
```

Each name is initialized to the lattice value $\top$. Propagation proceeds only along edges marked *executable*. Such marking takes place when the associated condition reaches a non-$\top$ value. The value $\top$ propagates along unexecutable edges.

SSA constant propagator (cont'd)

SSA Form

```
$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
    $i_2 \leftarrow \phi(i_1, i_5)$
    $j_2 \leftarrow \phi(j_1, j_3)$
    $k_2 \leftarrow \phi(k_1, k_4)$
    if $(i_2 = 6)$ then
        $k_3 \leftarrow 0$
    else
        $i_3 \leftarrow i_2 + 1$
    fi
    $i_4 \leftarrow \phi(i_2, i_3)$
    $k_4 \leftarrow \phi(k_3, k_2)$
    $i_5 \leftarrow i_4 + k_4$
    $j_3 \leftarrow j_2 + 1$
until $(i_5 = j_3)$
```

Pass 1

```
$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
    $i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge \top) = 6$
    $j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge \top) = 1$
    $k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = 1$
    if $(i_2 = 6)$ then
        $k_3 \leftarrow 0$
    else
        /⋆      Not executed      ⋆/
    fi
    $i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
    $k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
    $i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
    $j_3 \leftarrow j_2 + 1 \Rightarrow (1 + 1) = 2$
until $(i_5 = j_3 \Rightarrow (6 = 2) =$ false$)$
```

SSA constant propagator (cont'd)

Pass 1

```
$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
    $i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge \top) = 6$
    $j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge \top) = 1$
    $k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = 1$
    if $(i_2 = 6)$ then
        $k_3 \leftarrow 0$
    else
        /$\star$    Not executed    $\star$/
    fi
    $i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
    $k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
    $i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
    $j_3 \leftarrow j_2 + 1 \Rightarrow (1 + 1) = 2$
until $(i_5 = j_3 \Rightarrow (6 = 2) = $ false$)$
```

Pass 2

```
$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
    $i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge 6) = 6$
    $j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge 2) = \perp$
    $k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = \perp$
    if $(i_2 = 6)$ then
        $k_3 \leftarrow 0$
    else
        /$\star$    Not executed    $\star$/
    fi
    $i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
    $k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
    $i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
    $j_3 \leftarrow j_2 + 1 \Rightarrow (\perp + 1) = \perp$
until $(i_5 = j_3 \Rightarrow (6 = \perp) = \perp)$
```

Our solution has stabilized. Even though k_2 is $\perp$, that value is never transmitted along the unexecuted edge to the ϕ for k_4.

SSA value numbering [3, 39]

$a \leftarrow read()$
$v \leftarrow a + 2$
$c \leftarrow a$
$w \leftarrow c + 2$
$t \leftarrow a + 2$
$x \leftarrow t - 1$

For the above program, constant propagation will fail to determine a compile-time value for v and w, because the behavior of the $read()$ function must be captured as $\perp$ at compile-time.

Nonetheless, we can see that v and w will hold the same *value*, even though we cannot determine at compile-time exactly what that value will be. Such knowledge helps us replace the computation of $(c+2)$ by a simple copy from v.

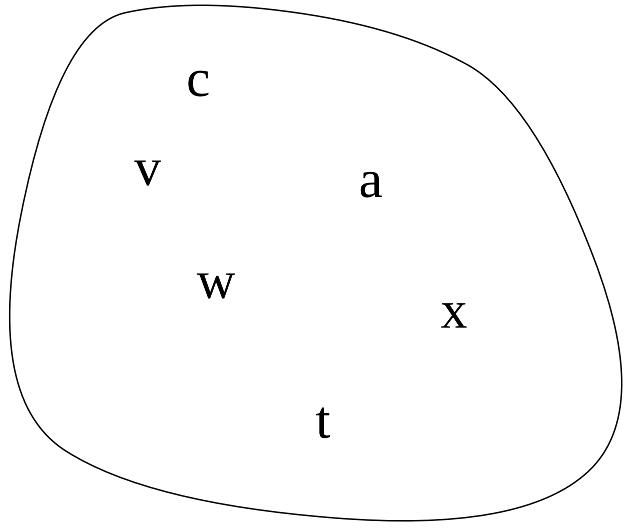
Value numbering attempts to label each computation of the program with a number, such that identical computations are identically labeled.

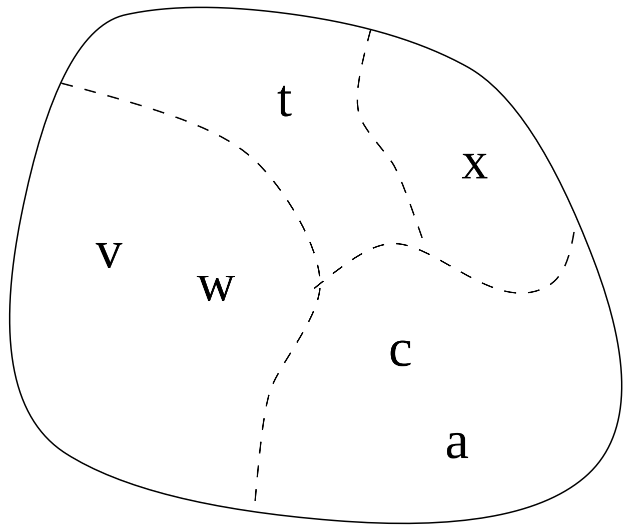
- **Prior to SSA form, value numbering algorithms were applied only within basic blocks (i.e., no branching) (2).**
- **Early value numbering algorithms relied on textual equivalence to determine value equivalence. The text of each expression (and perhaps subexpression) was *hashed* to a value number. Intervening defs of variables contained in an expression would kill the expression. This approach could not detect equivalence of v and w in the example to the left, since $(a+2)$ is not textually equivalent to $(c+2)$.**

It seems that x ought to have the same value as v and w, but our algorithm won't discover this, because the "function" that computes x ($\lambda n.n - 1$) differs from the "function" that computes v and w ($\lambda n.n + 2$).

SSA value numbering (cont'd)

- **We essentially seek a** *partition* **of SSA names by value equivalence, since value equivalence is reflexive, symmetric, and transitive.**
- **We'll initially assume that all SSA names have the same value.**
- **When evidence surfaces that a given block may contain disparate values (names), we'll talk about** *splitting* **the block. Generally, the algorithm will only split a block in two. However, the first split is more severe, in that names are split by the functional form of the expressions that compute their value.**



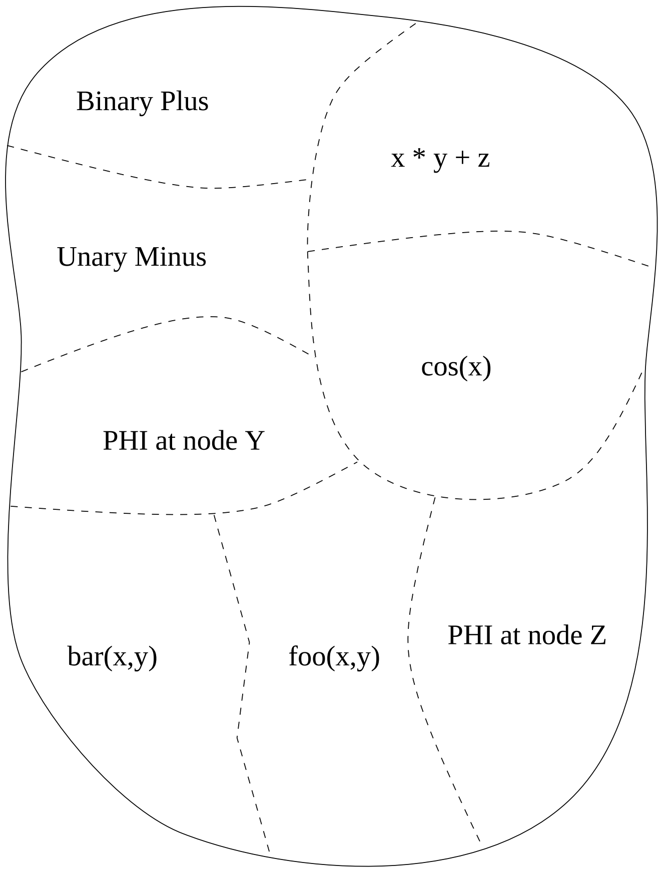


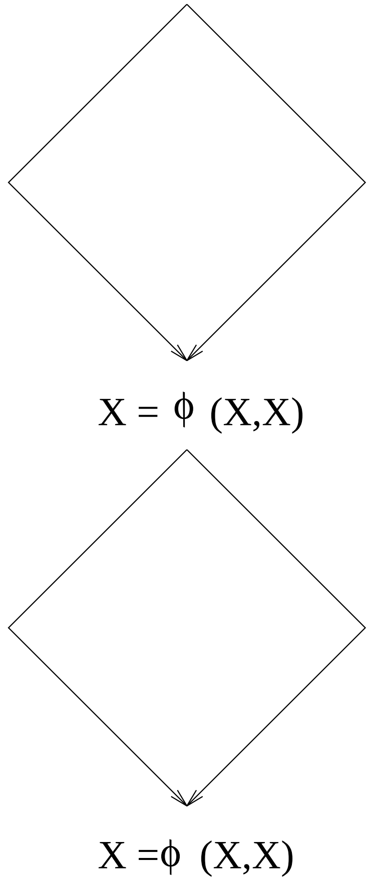
Above are shown the initial and final partitions for the example on the previous page.

SSA value numbering (cont'd)

After construction of SSA form, we split by the function name that computes values for the assigned variables. We thus distinguish between binary addition, multiplication, etc.

One further point is that ϕ-functions at different nodes must also be distinguished, even though their function form appears the same. This is necessary because a branch taken into one ϕ-function is not necessarily the same branch taken into another, unless the two functions reside in the same node.





SSA value numbering example

```
if $(condA)$ then
    $a_1 \leftarrow \alpha$
    if $(condB)$ then
        $b_1 \leftarrow \alpha$
    else
        $a_2 \leftarrow \beta$
        $b_2 \leftarrow \beta$
    fi
    $a_3 \leftarrow \phi(a_1, a_2)$
    $b_3 \leftarrow \phi(b_1, b_2)$
    $c_2 \leftarrow \star a_3$
    $d_2 \leftarrow \star b_3$
else
    $b_4 \leftarrow \gamma$
fi
$a_5 \leftarrow \phi(a_1, a_0)$
$b_5 \leftarrow \phi(b_0, b_4)$
$c_3 \leftarrow \star a_5$
$d_3 \leftarrow \star b_5$
$e_3 \leftarrow \star a_5$
```

For brevity, symbols α, β, and γ represent syntactically distinct function classes in the program shown to the left.

In the figures that follow, we'll see that c_2 and d_2 have the same value, while c_3 and d_3 do not. Thus, program optimization will save a memory fetch by using the value of c_2 for d_2.

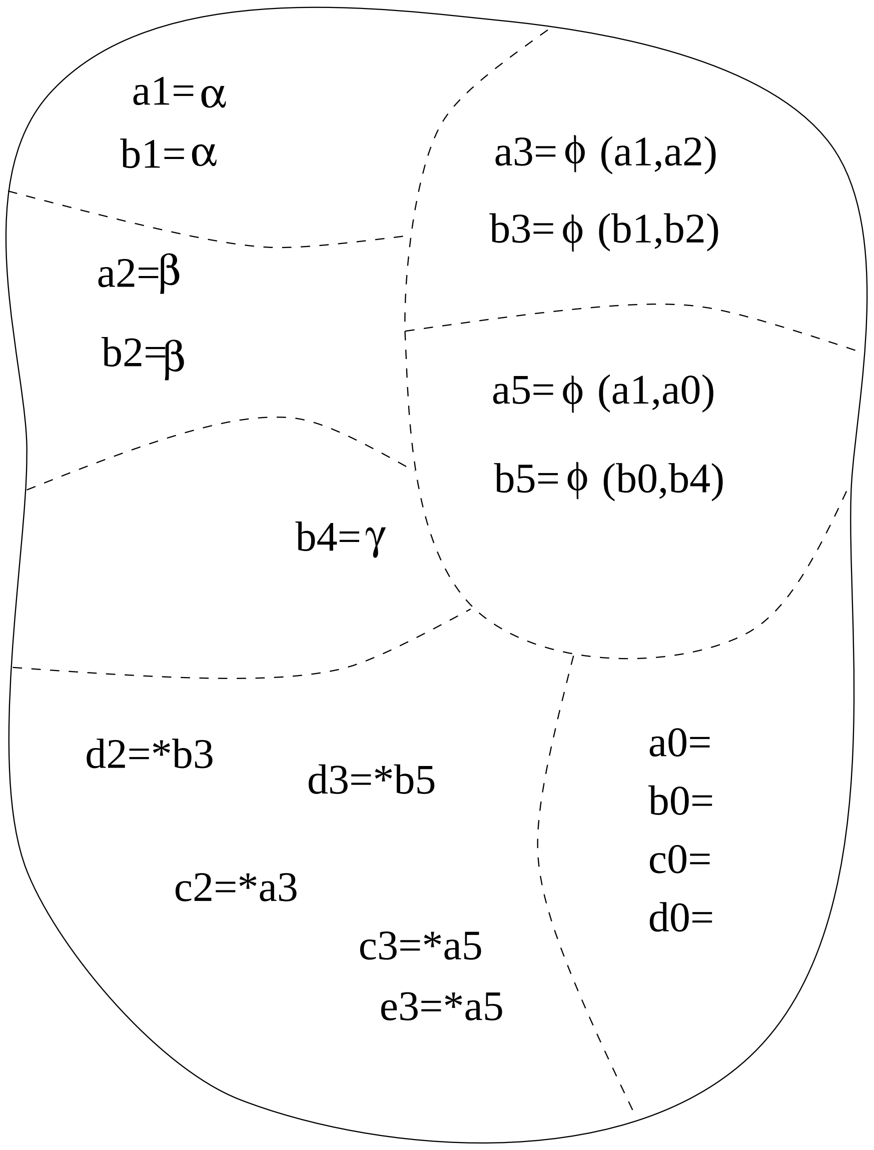
Note that if b is declared *volatile* in the language C, then this optimization would be incorrect, because each reference to b should be realized. How can one account for volatility in this optimization? Perhaps by assuming that volatile variables cannot have the same value.

It would be difficult and expensive to express all possible defs of a volatile variable in SSA form.

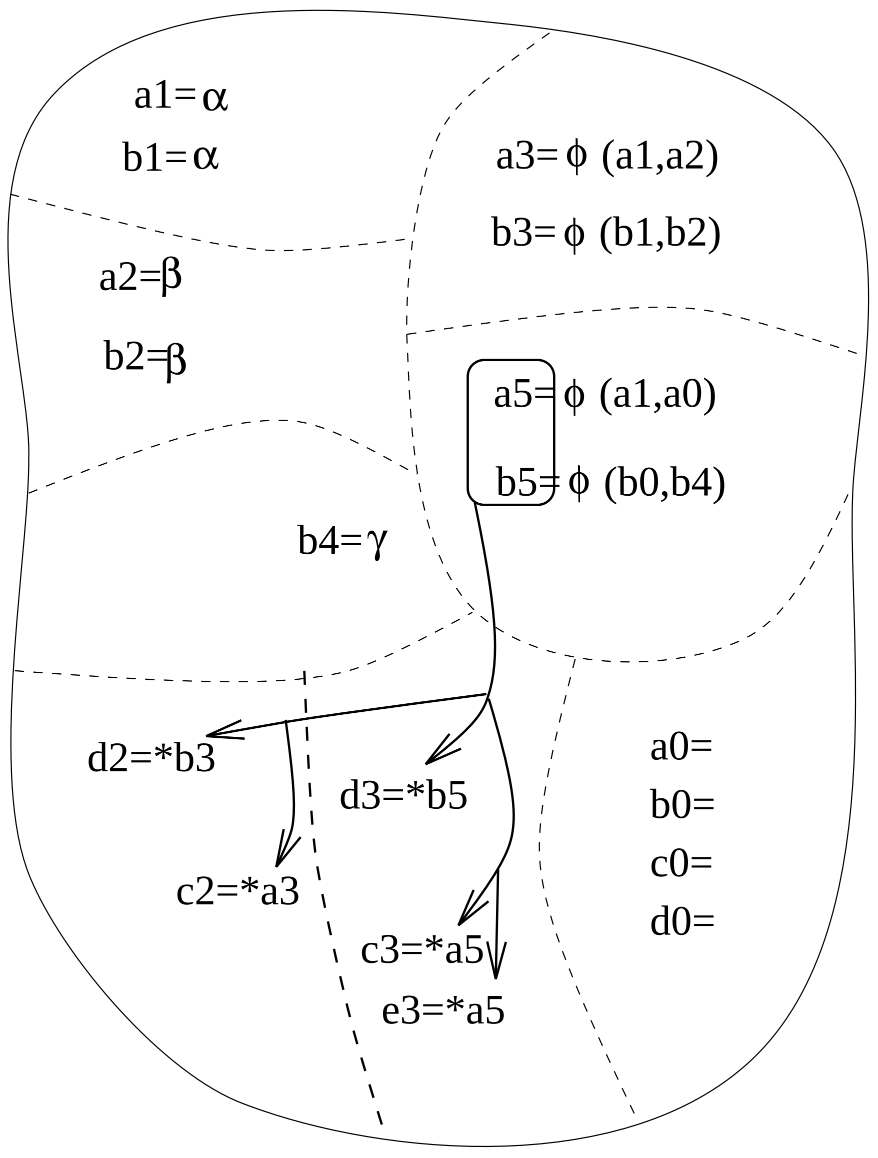
SSA value numbering example (cont'd)

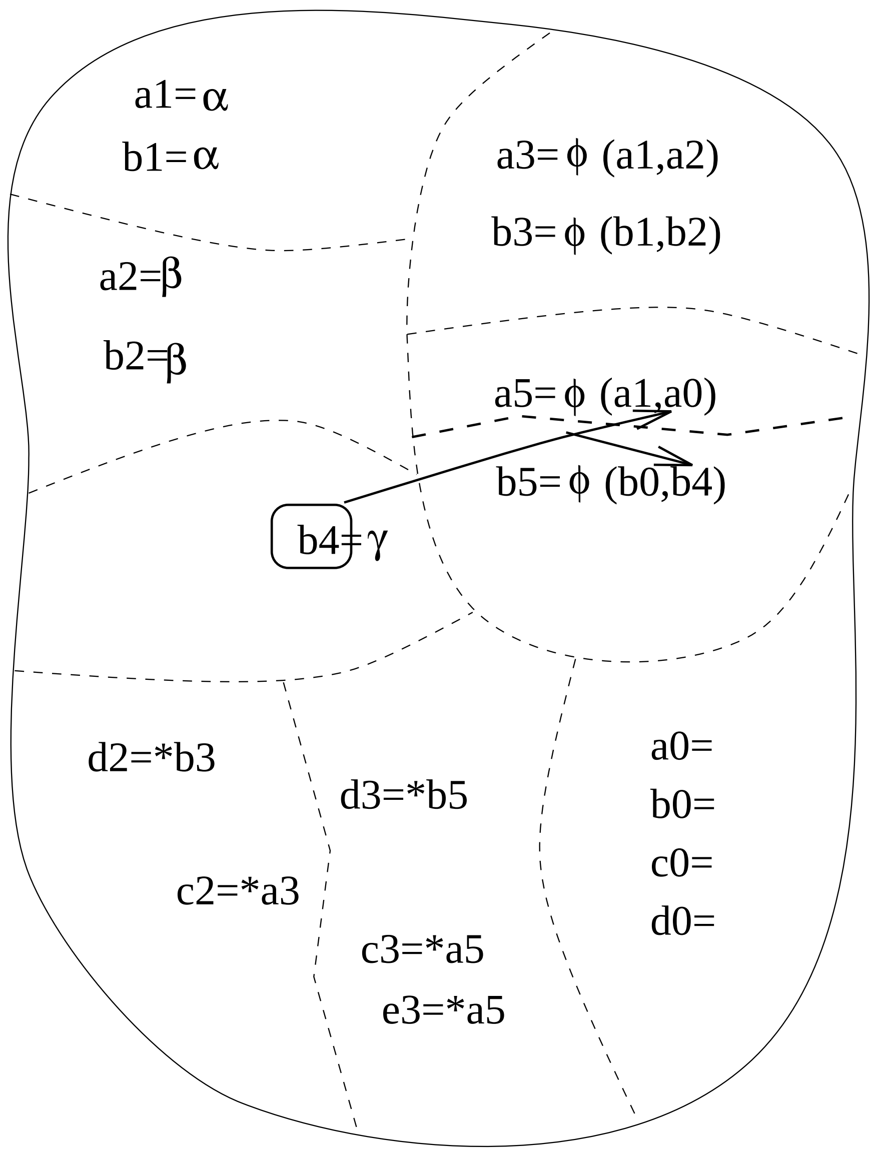
Here we see the initial partition of SSA names:

- **The syntactic classes α, β, and γ are distinguished;**
- **ϕ-functions at different nodes are distinguished;**
- **The initial value of each variable v_0 is considered identical;**
- **Within each syntactic class, values are considered identical.**



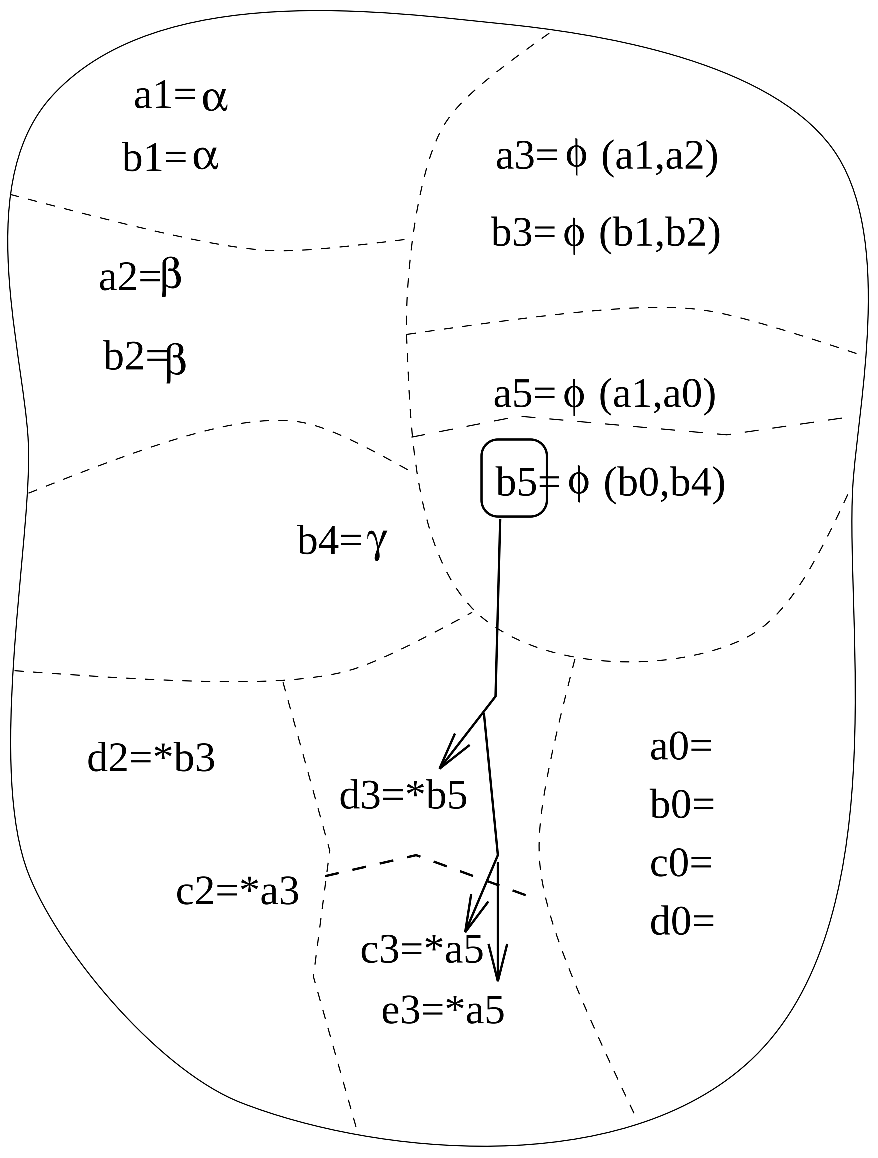
SSA value numbering example (cont'd)





On the left, the block with a_5 splits the five names shown into two subblocks; on the right, b_4 splits a_5 from b_5.

SSA value numbering example (cont'd)



Finally, b_5 splits c_3 from d_3. Here, note that we could have used either a_5 or b_5 to do the job. Asymptotic efficiency is gained by choosing b_5, because there are fewer uses of that name than of a_5.

In summary, the algorithm is as follows:

1. **Let W be a worklist of blocks to be used for further splitting.**
2. **Pick and remove (arbitrary) block D from W.**
3. **For each block C properly split by D,**
   - **(a) If C is on W, then remove C and enqueue its splits by D;**
   - **(b) Otherwise, enqueue the split with the fewest uses.**
4. **Loop to step 2 until W is empty.**

Register allocation

- Optimal register allocation is NP-hard.
- Trivial approaches can be really bad: using the most recently freed register is provably worst for pipelined machines.
- Many approaches begin by assuming an infinite number of *virtual registers* for assignment to values. These are then covered by actual registers during allocation.

Chaitin-Chandra

Each variable (or expression) is assigned a virtual register for the duration of a procedure. Actual registers are allocated by coloring an interference graph, using Chandra's heuristic. Where allocation fails, some expressions care chosen for *spilling*: these are not kept in registers but loaded on demand and immediately stored afterwards.

Chow-Hennessey

The maximum number of live variables is computed. Some register allocation can clearly succeed if there are sufficient registers to cover max-live. However, this may involve allocating the same variable to two different registers in different live ranges. This necessitates swapping registers for a given variable where control flow merges.

Knobe and Zadeck give a method that sloshes rather than spills: variables are kept intermittently in registers.

Possible course and project coverage

I've found that if one tries to cover all the parsing methods in sequence, then the projects tend to fall behind because the necessary lectures haven't been given. My solution is to alternate between the two kinds of lectures. I've tried to develop examples that serve as glue. One such example is the grammar that structures left and right values. While this is a good introduction to type checking, the grammar also illustrates the limitations of SLR parsing.

I begin with a few warm up assignments

1. The Chinese menu problem (10 days).
2. A finite-state machine problem, such as a reserved keyword or table generator (12 days).
3. Prefix expression evaluation, by recursive descent and a simple YACC grammar (2 weeks).

And then start the sequence of assignments that leads to a finished compiler.

1. Symbol tables, starting with the C grammar (2 weeks).
2. Abstract syntax trees (10 days).
3. Semantic analysis: left and right values, type checking (10 days).
4. Preliminary code generation: simple expressions (10 days).
5. Final code generation (2–3 weeks).

Conclusions

- **Much of the work in crafting a compiler has been automated, by parser generators, tokenizing tools, attribute grammars, and automatic code generators.**
- **Partly due to these tools, creating the runtime library now occupies a large portion of compiler construction time.**
- **To understand and better appreciate the automatic tools, I believe in giving students practice in creating some components "by hand".**
- **It's important to get all the way through code generation in a one-semester course.**
- **I prefer to leave program optimization for a second course.**
- **In a projects course of this nature, one is often loathe to assign (or grade) homework. I have found the following strategy works well:** [a] **Give out a list of 6 problems that the students should be able to work. One week later, give one of the six problems as a quiz, where the problem is determined by the roll of a die.**

[a] Thanks to Ken Goldman and Sally Goldman for this idea.

In summary

Sung to the tune:
"I am the very model
of a modern Major General"

We start with some descriptions of our languages fanatical
That specify the syntax and the attributes grammatical
Through Yacc and Lexx our BNF is processed quite dramatical
By front ends that we generate these parsers automatical.

They shift, reduce, and scrutinize our errors problematical
And sometimes honest programs get transformed into the radical
But what the heck we know our derivations are canonical
And you´ll admit our diagnostics are the most laconical.
And you'll admit our diagnostics are the most laconical,
Yes you'll admit our diagnostics are the most laconical,
Because we know our diagnostics are the most laconi conical.

Code motion, hoisting, commoning and all the transforms you'd expect
Your program's faster even if the output isn't quite correct.
But most of us believe our transformations are canonical
And you'll admit our diagnostics are the most laconical.

Some textbooks

Aho, Sethi, Ullman: *Compilers: principles, techniques, and tools*, **Addison-Wesley, 1988 (affectionately called "The Dragon Book").** A long-time favorite in classes and as a reference book. Good coverage of popular parsing methods, though Earley's method is not presented. Good description of runtime storage organization. No real connections are given to projects, and no tools are provided (but references to extant tools are given). No real insight given on how to disambiguate a grammar.

Fischer and LeBlanc: *Crafting a Compiler with* C, **Benjamin/Cummings, 1991.** There are actually two versions of this book, one with C and one based on ADA. This is an excellent textbook (my favorite), with excellent coverage of parsing, semantic analysis, and code generation. The text meshes nicely with tools provided by the authors.

Mason and Brown: *lex & yacc*, **O'Reilly & Associates, 1992.** A good companion for the tools it covers. I list this as an optional reference book.

Waite and Carter: *An Introduction to Compiler Construction*, **HarperCollins, 1993.** A relatively new book, strong on code generation, but thin in parsing and semantic analysis. Biased toward the VAX instruction set. A consistent well-integrated text. Very weak on grammars: the text uses syntax diagrams.

Waite and Goos: *Compiler Construction*, **Springer-Verlag, 1984.** A now-dated but good text, fairly broad and not overly deep in any one area.

My favorite in teaching has been Fischer and LeBlanc. That book is currently undergoing revision, which will merge the language-specific versions and include much new material, for example on program optimization. I am joining them as a coauthor in this revision.

Reference books

Aho and Ullman: *The Theory of Parsing, Translation, and Compiling* **(two volumes), Prentice-Hall, 1973.** Not really a textbook, but an excellent reference.

Bauer and Eickel: *Compiler Construction: An Advanced Course*, **Springer-Verlag, 1976.** Notes from a course taught in 1974. A good teaching reference, but not a textbook. The chapters are separately authored.

Hopcroft and Ullman: *Introduction to Automata Theory, Languages, and Computation*, **Addison-Wesley, 1979.** A formal text on language recognition. A good reference for teaching.

Martin: *Introduction to Languages and the Theory of Computation*, **McGraw–Hill, 1991.** An excellent reference on automata theory. Terrific coverage of undecidability.

Wirth: *Algorithms + Data Structures = Programs*, **Prentice-Hall, 1976.** Really a book on other topics, but includes great coverage of recursive-descent compilers for PASCAL. Also uses syntax diagrams.

References

(1) Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling, Volume II*. Prentice-Hall, Inc., 1973.

(2) A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

(3) B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 1–11, January 1988.

(4) Andrew W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation*, chapter 4. The MIT Press, 1991.

(5) M. Auslander and M. Hopkins. An overview of the PL.8 compiler. *Proc. SIGPLAN'82 Symp. on Compiler Construction*, pages 22–31, June 1982. Published as *SIGPLAN Notices* Vol. 17, No. 6.

(6) R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand driven interpretation of languages. *Proc. SIGPLAN'90 Symp. on Compiler Construction*, pages 257–271, June 1990. Published as *SIGPLAN Notices* Vol. 25, No. 6.

(7) Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the rn programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

(8) R. Cytron, A. Lowry, and F. K. Zadeck. Code motion of control structures in high-level languages. *Conf. Rec. Thirteenth ACM Symp. on Principles of Programming Languages*, pages 70–85, January 1986.

(9) Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, October 1991.

(10) Ron K. Cytron and David Shields. FRIL – a fractal intermediate language. Technical report, Washington University in St. Louis, 1993. Report number WUCS 93-51 from Washington University in St. Louis and wuarchive ftp.

(11) J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *ACM Trans. on Programming Languages and Systems*, 6(4):505–526, October 1984.

(12) D. M. Dhamdhere. Practical adaptation of the global optimization algorithm of morel and renvoise. *ACM Trans. on Programming Languages and Systems*, 13(2):291–294, April 1991.

(13) D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyse large programs efficiently and informatively. *Proc. SIGPLAN'92 Symp. on Compiler Construction*, June 1992. to appear.

(14) Dhananjay M. Dhamdhere and Uday P. Khedker. Complexity of bi-directional data flow analysis. *Conf. Rec. Twentieth ACM Symp. on Principles of Programming Languages*, 1993.

(15) J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

(16) Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a Compiler with C*. Benjamin/Cummings Publishing Company, Inc., 1991.

(17) C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

(18) C.W. Fraser and D.R. Hanson. A retargetable compiler for ansi c. *SIGPLAN Notices*, 26(10):29–43, 1991.

(19) C.W. Fraser, R.R. Henry, and T.A. Proebsting. Burg–fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, 1992.

(20) Mahadevan Ganapathi and Charles N. Fischer. Affix grammar driven code generation. *ACM Transactions on Programming Languages and Systems*, 7(4):560–599, October 1985.

(21) R. S. Glanville and S. L. Graham. A new method for compiler code generation. *Conf. Rec. Fifth ACM Symp. on Principles of Programming Languages*, pages 231–240, January 1978.

(22) L. H. Holley and B. K. Rosen. Qualified data flow problems. *IEEE Trans. on Software Engineering*, SE-7(1):60–78, January 1981.

(23) Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. *SIGPLAN Notices*, 27(7), 1992. SIGPLAN PLDI Conference Proceedings.

(24) J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

(25) J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

(26) M.E. Lesk and E. Schmidt. Lex–a lexical analyzer generator. In *UNIX Programmer's Manual 2*. AT&T Bell Laboratories, 1975.

(27) T. J. Marlowe. *Data Flow Analysis and Incremental Iteration*. PhD thesis, Dept. of Computer Science, Rutgers U., October 1989.

(28) John C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw–Hill, Inc., 1991.

(29) Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *Software Engineering Notes*, 9(3), 1984. Also appears in Proceedings of the ACM Symposium on Practical Programming Development Environments, Pittsburgh, PA, April, 1984, and in SIGPLAN Notices, Vol. 19, No 5, May, 1984.

(30) David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 1994.

(31) T. W. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer-Verlag, NY, NY, 1989.

(32) T. W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual*. Springer-Verlag, NY, NY, 3rd edition, 1989.

(33) Dennis Ritchie. A tour through the unix c compiler. Technical report, AT&T Bell Laboratories, 1979.

(34) B. K. Rosen. High level data flow analysis. *Comm. ACM*, 20(10):712–724, October 1977.

(35) B. K. Rosen. Data flow analysis for procedural languages. *J. ACM*, 26(2):322–344, April 1979.

(36) B. K. Rosen. Monoids for rapid data flow analysis. *SIAM J. Computing*, 9(1):159–196, February 1980.

(37) B. K. Rosen. Degrees of availability as an introduction to the general theory of data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*, chapter 2, pages 55–76. Prentice Hall, 1981.

(38) B. K. Rosen. A lubricant for data flow analysis. *SIAM J. Computing*, 11(3):493–511, August 1982.

(39) B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 12–27, January 1988.

(40) Vivek Sarkar. Determining average program execution times and their variance. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 24(7):298–312, July 1989.

(41) R. T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: a syntax-directed programming environment. *Comm. ACM*, 24(9):563–573, September 1981.

(42) G. A. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis techniques. *Proc. SIGPLAN'89 Symp. on Compiler Construction*, pages 1–12, 1989. Published as *SIGPLAN Notices* Vol. 24, No. 7.

(43) Jr. Vincent A. Guarna, Dennis Gannon, David Jablonowski, Allen D. Malony, and Yogesh Gaur. Faust: An integrated environment for the development of parallel programs. Technical report, U. of Il.-Center for Supercomputing Research and Development, November 1988. CSRD Rpt. No. 825 to appear in IEEE Software, 1989.

(44) M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, April 1991.

(45) P. T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs*. PhD thesis, Xerox Palo Alto Research Center, Palo Alto, Ca. 94304, May 1984.