

Revision Notes

What is a Compiler?

Compilers are program translators, e.g., C → x86 assembler.

The source language can be anything from high-level programming languages and modelling languages to document description languages and database query languages. Target languages could be another high-level programming language, a low-level programming language (assembler or machine code) or an application-specific target language.

Compilers vs. Interpreters

Interpreters are another class of translators. Whereas a compiler translates a program once and for all into target language, an interpreter effectively translates (the used parts of) a source program every time it is run. Techniques like Just-In-Time Compilation (JIT) blurs this distinction. Compilers and interpreters are sometimes used together, e.g. in Java, the source code is compiled into Java byte code and the byte code interpreted by a Java Virtual Machine (JVM).

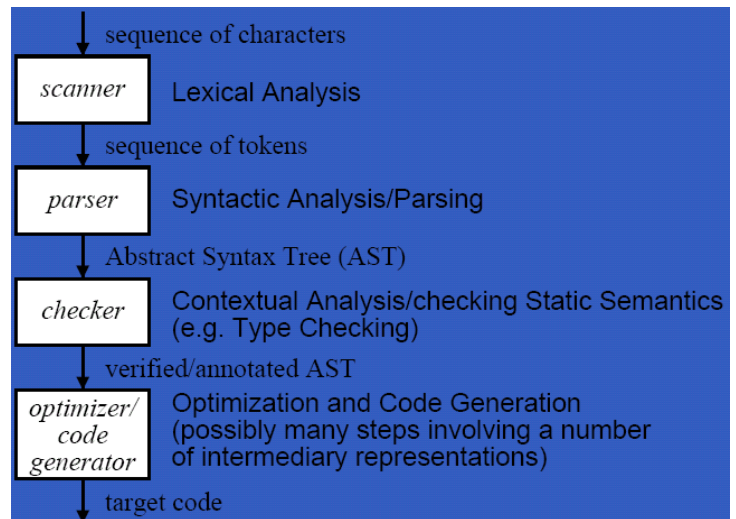
Traditionally, compilation is broken down into several steps:

- Lexical Analysis:
 - Verify that input character sequence is lexically valid (i.e. in the language).
 - Group characters into sequence of lexical symbols - tokens.
 - Discard white space and comments (typically).

- Syntactic Analysis/Parsing
 - Verify that the input program is syntactically valid, i.e. conforms to the Context Free Grammar of the language.
 - Determine the program structure.
 - Construct a representation of the program reflecting that structure without unnecessary details, usually an Abstract Syntax Tree (AST).

- Contextual Analysis/Checking Static Semantics:
 - Resolve meaning of symbols.
 - Report undefined symbols.
 - Type checking.

- Optimization and Code Generation:
 - Code improvements aiming at making it run faster and/or use less space.
 - Output the appropriate sequence of target language instructions.



The 'Front-end' of a compiler, i.e. the scanner, parser, contextual checker, etc. depend more heavily on the source language, whereas the 'back-end' (optimizer, code generator, etc.) depends more heavily on the target language. The 'middle-end' can refer to parts that operate on a intermediary representation that is fairly independent of source and target language and hence reusable.



Example – Trivial eXpression Language (TXL):

Some examples of TXL programs (concrete syntax) and their intended meaning (semantics):

1 + 3	Meaning: 4
1 + 3 * 4	Meaning: 13
let x = 3 * 7 in x + 3	Meaning: 24
let x = 3 * 7 in let x = x * 3 in x - 21	Meaning: 42

- The only significance of white space is to separate tokens.
- Comments starts with ! and runs to the end of the line.
- Comments are treated as white space.
- TXL has only a single integer type, so typing is trivial.
- Only defined variables may be used, i.e. let x = 3 in y + x is invalid.

Context Free Grammar for TXL

The formal Context Free Grammar (CFG) for TXL in Backus-Naur Form (BNF):

```

txl-program  →   exp
exp          →   add-exp
add-exp     →   mul-exp
              | add-exp + mul-exp
              | add-exp - mul-exp
mul-exp     →   prim-exp
              | mul-exp * prim-exp
              | mul-exp / prim-exp
prim-exp    →   INTEGER
              | IDENTIFIER
              | ( exp )
              | let IDENTIFIER = exp in exp

```

Some Useful Terms and Definitions

Syntax: the form of programs

Semantics: the meaning of programs

Context Free Grammar (CFG): formal way of describing syntax

Backus-Naur Form (BNF): a particular way of expressing CFGs

Concrete Syntax: the exact character sequences that are syntactically valid programs

Derivation (or Parse) Tree: tree-structured record of the derivation of a terminal string from a CFG

Phrase Structure: the essential structure of syntactically valid programs, i.e. the way in which a program is constructed from phrases and sub-phrases.

Abstract Syntax: concrete representation of the phrase structure.

Abstract Syntax Tree: a simplified parse tree, concrete representation of the phrase structure for some particular sentence

Object Code: target program, especially when in machine code of some form

Token or Lexeme: basic language symbol, like integers, identifiers, keywords, parentheses

Lexical Analysis: determining the lexical structure of the source program (as a character sequence), breaking it into tokens

Scanner: carries out the lexical analysis

Parsing: determining the phrase structure of the source program (as a token sequence)

Parser: carries out the parsing, usually also constructs an AST (if the input is syntactically valid)

Static Semantics or Contextual Constraints: aspects of well-formedness that are to be checked *statically* (at compile time), but cannot be expressed by a *context-free* grammar. For example, scope rules, type rules.

Contextual Analysis: verifying that a program respects the static semantics

Optimizer: transforms the internal representation of a program to make the final target program run faster and/or use less space

Specifying Programming Languages

Before we start designing a compiler, we must understand the source language; i.e. the Syntax and Semantics of the Source Language must be specified - programmers using a compiler need to understand the source language and the source language syntax must be known to design the scanner and parser properly. Ultimately, the *target language* must be similarly specified in order to generate syntactically correct target code. Specifications can be *formal* or *informal*. Usually they are somewhere in between. Knowing the semantics of both source and target language preserves the meaning of programs through the compile process. Also, if we want to prove a compiler correct, the first step is the formal specification of both the source and target language.

In any language specification, informal or formal, a careful distinction must be made between the object language - the language being specified - and the meta language - the language of the specification itself. Moreover, the semantics of the meta language must be well understood.

A specification is formal if a formal meta language is used. A meta language is formal if it has a well-defined mathematical meaning. Certain well-defined, high-level programming languages, like Haskell, can arguably be thought of as formal specification languages, allowing for executable specifications. EBNF is a formal meta language for specifying context-free syntax. Type systems are increasingly specified formally using inference rules and logic. There are various approaches for specifying dynamic semantics formally. While most programming languages tend to be defined more or less informally, there are some formally defined languages; e.g. Algol 68 (using the Vienna Definition Method) and Standard ML (a functional language like Haskell).

In an informal specification, the meta language is a natural language such as English. Note that "Informal" does not imply "lack of rigour" - it is possible to be precise in a natural language. An example of a well-written, predominantly informal language specification is that of Java (<http://java.sun.com/docs/books/jls>).

A formal specification allows important properties to be proved about the specification. For example, type soundness for a type system. This gives additional levels of assurance regarding a language design (an early version of Java's type system was flawed). A good formal specification is accompanied by informal explanations and examples.

The MiniTriangle Language

MiniTriangle Lexical Grammar

Program	→	(Token Separator)*
Token	→	Keyword Identifier Integer-Literal Operator ; : := ~ () eot
Keyword	→	begin const do else end if in let then var while
Identifier	→	Letter Identifier Letter Identifier Digit <i>but it must not be a Keyword</i>
Integer-Literal	→	Digit Integer-Literal Digit
Operator	→	+ - * / < > = \
Separator	→	Comment space eol
Comment	→	Graphic* eol

Some valid MiniTriangle tokens include const3 (Identifier), const (Keyword), 42 (Integer-Literal), + (Operator). Why is const3 a single token and not two? The grammar is ambiguous. We get around this problem by using the so-called 'maximal munch rule' - we always try and form the longest possible token.



Revision Notes

MiniTriangle CFG

Program	→	single-Command
Command	→	single-Command Command ; single-Command
single-Command	→	Identifier := Expression Identifier (Expression) if Expression then single-Command else single-Command while Expression do single-Command let Declaration in single-Command begin Command end
Expression	→	primary-Expression Expression Operator primary-Expression
primary-Expression	→	Integer-Literal Identifier Operator primary-Expression (Expression)
Declaration	→	single-Declaration Declaration ; single-Declaration
single-Declaration	→	const Identifier ~ Expression var Identifier : Type-denoter
Type-denoter	→	Identifier

The given lexical grammar is expressed in BNF and looks pretty similar to the context free grammar. Why not join them, do away with scanning, and just do parsing?

- Simplicity: dealing with white space and comments in the context free grammar becomes extremely complicated.
- Efficiency: Working on classified groups of characters (tokens) facilitates parsing. It may be possible to use a simpler parsing algorithm.
- Grouping and classifying characters by as simple means as possible increases efficiency.

The MiniTriangle lexical grammar is a regular grammar since it is left linear (recursive calls are always the left most option). Thus it could have been expressed using a less powerful formalism such as regular expressions. Regular languages can be recognised more simply (by a Deterministic Finite state Automaton, DFA) and thus more efficiently than context-free languages.

MiniTriangle Abstract Syntax Grammar

This grammar specifies only the *phrase structure* of MiniTriangle. In addition, it gives a textual representation for ASTs, as well as node labels to be used when drawing ASTs as trees.

Program	→	Command	<i>Program</i>
Command	→	Identifier := Expression Identifier (Expression) Command ; Command if Expression then Command else Command while Expression do Command let Declaration in Command	<i>AssignCommand</i> <i>CallCommand</i> <i>SequentialCommand</i> <i>IfCommand</i> <i>WhileCommand</i> <i>LetCommand</i>
Expression	→	Integer-Literal Identifier Operator Expression Expression Operator Expression	<i>IntegerExpression</i> <i>IdentifierExpression</i> <i>UnaryExpression</i> <i>BinaryExpression</i>
Declaration	→	const Identifier ~ Expression var Identifier : Type-denoter Declaration ; Declaration	<i>ConstDeclaration</i> <i>VarDeclaration</i> <i>SequentialDeclaration</i>
Type-denoter	→	Identifier	<i>SimpleTypeDenoter</i>

MiniTriangle is very simple - it insists on all used identifiers being declared. The scope of a declaration is the body of the let except where shadowed by a redeclaration. However, in more complex languages, the important question about *static vs. dynamic binding* arises.

The issue is deciding which declaration (or *binding occurrence*) corresponds to which use (or *applied occurrence*). A language exhibits *static binding* if this can be determined statically, at compile-time, without running a program or *dynamic binding* if this only can be determined dynamically, at run-time, by

Revision Notes

actually running a program. Every programming language has a ‘universe of discourse’. The elements of this are called *values*. The set of possible *values* is usually classified into *types*. All operations in a language have associated *type rules* that tell the expected type of operands and the result type (if any). Any application that does not satisfy the associated type rule is a *type error*.

A programming language is *statically typed* if all type errors can be found statically, at compile-time, without running a program or *dynamically typed* if type errors can only be found dynamically, at run-time, by actually running a program.

Mini triangle is statically typed. Some type rules:

>: If both operands are of type int, then the result is of type bool.

while E do C: E must be of type bool

Syntax Analysis

A *symbol* is a basic indivisible entity. Concrete examples of symbols are letters and digits. A *string* or *word* is a finite sequence of juxtapositioned symbols. An *alphabet* is a finite set of symbols. ϵ denotes the word of length 0, the *empty word*. A *language* (over alphabet Σ) is a set of words (over alphabet Σ). Σ^* denotes the set of *all* words over an alphabet Σ , including ϵ .

Concatenation of words is denoted by juxtaposition, i.e., Concatenation of ab and ba yields abba.

Concatenation is associative and has unit ϵ , i.e., $u(vw) = (uv)w$ and $\epsilon u = u = u\epsilon$

Concatenation of words is extended to languages by $MN = \{uv \mid u \in M \wedge v \in N\}$

Concatenation of languages is associative and has unit $\{\epsilon\}$, i.e., $L(MN) = (LM)N$ and $L\{\epsilon\} = L = \{\epsilon\}L$

Concatenation distributes through set union: $L(M \cup N) = LM \cup LN$ and $(L \cup M)N = LN \cup MN$

A *Context-Free Grammar* (CFG) is a way of formally describing *Context-Free Languages* (CFL):

The CFLs captures ideas common in programming languages such as

- nested structure
- balanced parentheses
- matching keywords like begin and end.

Most “reasonable” CFLs can be recognised by a fairly simple machine: a *deterministic pushdown automaton*.

Describing a programming language by a “reasonable” CFG allows context-free constraints to be expressed as well as imparting a hierarchical structure to the words in the language. It allows simple and efficient *parsing* (i.e., determining if a word belongs to the language and if so, determining its *phrase structure*).

A *Context-Free Grammar* is a 4-tuple (N, T, P, S) where

N is a finite set of *nonterminals*, T is a finite set of *terminals* (the *alphabet* of the language being described), $N \cap T = \emptyset$ (N and T are disjoint), S , the *start symbol*, is a distinguished element of N and P is a finite set of productions, written $A ::= \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$

Productions with the same LHS are usually grouped together.

In order to formally define the language generated by a grammar $G = (N, T, P, S)$, we first define a binary relation \Rightarrow^G on strings over $N \cup T$, read “directly derives in grammar G ”, being the least relation such that $\alpha A \gamma \Rightarrow^G \alpha B \gamma$ whenever $A \rightarrow B$ is a production in grammar G .

Example: Given the grammar

$S \rightarrow \epsilon \mid aA$

$A \rightarrow bS$

We have $S \Rightarrow \epsilon$, $S \Rightarrow aA$, $aA \Rightarrow abS$, $SaAaa \Rightarrow SabSaa$



The relation $*\Rightarrow^G$, read “derives in grammar G” is the reflexive, transitive closure of \Rightarrow^G , that is, $*\Rightarrow^G$ is the least relation on strings over $N \cup T$ such that:

$\alpha * \Rightarrow^G \beta$ if $\alpha \Rightarrow^G \beta$

$\alpha * \Rightarrow^G \alpha$ (reflexivity)

$\alpha * \Rightarrow^G \beta$ if $\alpha \Rightarrow^G \gamma \wedge \gamma * \Rightarrow^G \beta$ (transitivity)

Example: Given the grammar

$S \rightarrow \varepsilon \mid aA$

$A \rightarrow bS$

We have $S * \Rightarrow \varepsilon$, $S * \Rightarrow aA$, $aA * \Rightarrow abS$, $S * \Rightarrow abS$, $S * \Rightarrow ababS$, $S * \Rightarrow abab$

The language generated by a context-free grammar $G = (N, T, P, S)$ denoted $L(G)$, is defined as $L(G) = \{ w \mid w \in T^* \wedge S * \Rightarrow w \}$. A language L is a *Context-Free Language* (CFL) iff $L = L(G)$ for some CFG G . A string $\alpha \in (N \cup T)^*$ is a *sentential form* iff $S * \Rightarrow \alpha$.

Two grammars G_1 and G_2 are *equivalent* iff $L(G_1) = L(G_2)$. Note that the equivalence of CFGs is in general *undecidable*.

Regular Languages are a proper subset of context-free languages. Context-free grammars can thus be used to describe regular languages. In particular, if the form of the productions in a grammar G is restricted to make the grammar either *left-linear* or *right-linear*, then $L(G)$ is guaranteed to be a regular language. Thus, a right- or left-linear grammar is called a *regular grammar*.

A CFG $G = (N, T, P, S)$ is *right-linear* if all its productions are of the forms

$A ::= wB$

$A ::= w$

(i.e., the non-terminals are on the right)

Conversely, a CFG is *left-linear* if all its productions are of the forms

$A ::= Bw$

$A ::= w$

The MiniTriangle lexical grammar as presented earlier is (essentially) left-linear. The MiniTriangle lexical grammar is thus a *regular grammar* and the described language is consequently regular as well. This allows the MiniTriangle scanner to be implemented as a DFA, ensuring that lexical analysis is simple and efficient.

A tree is a *derivation* or *parse tree* for CFG $G = (N, T, P, S)$ if:

- every vertex has a *label* from $N \cup T \cup \{ \varepsilon \}$
- the label of the root is S
- labels of interior vertices belong to N
- if vertex n has label A and vertices n_1, n_2, \dots, n_k are the children of n , from left to right, with labels X_1, X_2, \dots, X_k , then $A \rightarrow X_1 X_2 \dots X_k$ is a production in P
- if a vertex n has label ε , then n is a leaf and the only child of its parent.

Given a derivation tree for a grammar G :

- the string of leaf labels read from left to right is the *yield* of the tree.
- the yield is a sentential form of G .
- a string α is the yield of some derivation tree for a grammar G iff $S * \Rightarrow^G \alpha$

A derivation is *leftmost* if productions are always applied to the leftmost nonterminal at each step in a derivation. A derivation is *rightmost* if productions are always applied to the rightmost nonterminal at each step in a derivation.



A CFG, G , is *ambiguous* if some word in $L(G)$ has *more than one leftmost derivation*, or *more than one rightmost derivation*.

A CFL for which every CFG is ambiguous is *inherently ambiguous*. Most CFLs are not inherently ambiguous; i.e., an ambiguous CFG G for a language L can often be *transformed* into an *equivalent* but unambiguous grammar G' . The ambiguity of a CFG is in general *undecidable*.

Eliminating Ambiguity

Example: Dangling-Else

Consider the following “dangling-else” grammar:

```
stmt  → if expr then stmt
      | if expr then stmt else stmt
      | other
```

and the following program fragment:

```
if expr1 then if expr2 then stmt1 else stmt2
```

Two possible parse trees. Hence the grammar is ambiguous.

Preferred interpretation: “Match each else with the closest previous unmatched then”

To accomplish this, we transform the grammar into an *equivalent* but *unambiguous* grammar.

```
stmt      → matched-stmt
          | unmatched-stmt
matched-stmt → if expr then matched-stmt else matched-stmt
          | other
unmatched-stmt → if expr then stmt
              | if expr then matched-stmt else unmatched-stmt
```

It is standard practice to leave out unnecessary parentheses when writing down mathematical expressions. One would normally expect to be able to do the same when writing programs.

For example:

$1 + 2 + 3$ instead of $(1 + 2) + 3$
 $47 - 3 - 2$ instead of $(47 - 3) - 2$

The following grammar achieves that:

```
expr  → integer
      | expr + expr
      | expr - expr
      | ( expr )
```

but is ambiguous.

To disambiguate, we want to make both $+$ and $-$ *left associative*. That can be achieved by making the relevant grammar productions *left recursive*:

```
expr      → prim-expr
          | expr + prim-expr
          | expr - prim-expr
prim-expr → integer
          | ( expr )
```

Other operators are usually *right associative*. For example, consider an arithmetic power operator \wedge . We would like $3 \wedge 2 \wedge 3$ to be evaluated as $3 \wedge (2 \wedge 3)$

An operator can be made *right associative* through *right recursive* grammar productions:

```
pow-expr  → prim-expr
          | prim-expr ^ pow-expr
prim-expr → integer
          | ( expr )
```

Revision Notes

One would also expect to be able to leave out parentheses when standard rules for *operator precedence* ought to make it clear what is meant. For example, it should be possible to write $1 + 2 * 3$ instead of having to write out the fully parenthesised version $1 + (2 * 3)$.

We chose to make $*$ left associative (standard). The following grammar accepts expressions like $1 + 2 * 3$:

```

expr      → prim-expr
          | expr + prim-expr
          | expr * prim-expr
prim-expr → integer
          | ( expr )
  
```

However, the meaning is *not* what we want! $1 + 2 * 3$ gets parsed as $(1 + 2) * 3$.

We rewrite the grammar so that expressions involving *high-precedence* operators only can occur as *subexpressions* of expressions involving low-precedence operators:

```

expr      → mul-expr
          | expr + mul-expr
mul-expr  → prim-expr
          | mul-expr * prim-expr
prim-expr → integer
          | ( expr )
  
```

Transforming a grammar to eliminate ambiguity is not always desirable: It can be quite hard to do correctly and the transformed grammar might be less easy to understand than the original. *Parser generator tools* often provide alternative disambiguation mechanisms such as meta-rules that favours the longest RHS among a group of conflicting productions or explicit declaration of operator precedence.

Extended BNF (EBNF) is a more convenient way of describing CFGs than BNF. EBNF uses:

- parentheses for grouping
- | for alternatives within parentheses
- * for iteration

Note that EBNF is no more powerful than BNF – the languages described are still the CFLs, and any EBNF grammar can be transformed into BNF.

EBNF to BNF

Grouping can be eliminated by introducing a new nonterminal for each group:

$A \rightarrow B (C | D | E) F$ is equivalent to $A \rightarrow BA_1F$
 $A_1 \rightarrow C | D | E$

The iterative construct can be replaced by explicit recursion:

$S \rightarrow a(bb)^*c$ generates the language $L(G) = \{a(bb)^i c \mid i \geq 0\}$

An equivalent left-recursive grammar is $S \rightarrow aAc$
 $A \rightarrow \varepsilon \mid Abb$

An equivalent right-recursive grammar is $S \rightarrow aAc$
 $A \rightarrow \varepsilon \mid bbA$

The EBNF grammar $\text{block} \rightarrow \text{begin} (\text{decl} \mid \text{stmt})^* \text{end}$ is equivalent to this BNF grammar:

```

block → begin block-rec end
block-rec → ε | block-rec block-alts
block-alts → decl | stmt
  
```

EBNF can evidently be quite a bit more concise and readable than plain BNF.



Revision Notes

Watt & Brown use their own EBNF variant. The more common variant is the ISO version (ISO/IEC 14977:1996) which uses curly braces (“{” and “}”) used to denote iteration (zero, one or more) and square brackets (“[” and “]”) used to denote options (zero or one).

If we use EBNF, we can substitute the RHS of a production for uses of the nonterminal it defines, as long as *all alternatives are included*:

$A \rightarrow X B Y$

$B \rightarrow C \mid D$

$B \rightarrow E$

can be transformed into

$A \rightarrow X (C \mid D \mid E) Y$

$B \rightarrow C \mid D$

$B \rightarrow E$

If all uses of a nonterminal are eliminated through substitution, all productions for that non-terminal can be deleted as well:

$A ::= X B Y$

$B ::= C \mid D$

$B ::= E$

can be transformed into $A \rightarrow X (C \mid D \mid E) Y$

Left Factoring

If we use EBNF, a common prefix among a group of productions can be factored out. Consider:

$A \rightarrow XYX$

$\mid XYZZY$

After left factoring:

$A \rightarrow XY (X \mid ZZY)$

Elimination of Left Recursion

Certain kinds of parsers cannot handle left-recursive productions. If it is desired to use such a parser, but the grammar is left-recursive, then the grammar first has to be transformed into an equivalent grammar that is not left-recursive.

We will first see how that can be done for *immediate* left recursion; i.e., productions of the form $A \rightarrow A\alpha$ (where α is not ϵ)

For each nonterminal A defined by some left-recursive production, group the productions for A :

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ such that no β_i begins with an A .

Then replace the A productions by

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A'$

We assume no α_i is ϵ .

To eliminate *general* left recursion, first transform the grammar into an *immediately* left-recursive grammar through systematic substitution then proceed as before.

For example, the generally left-recursive grammar

$A \rightarrow Ba$

$B \rightarrow Ab \mid Ac \mid \epsilon$

is first transformed into the immediately left-recursive grammar

$A \rightarrow Aba$

$A \rightarrow Aca$

$A \rightarrow a$

Revision Notes

There are two basic strategies for parsing - *top-down* and *bottom-up*. We will initially focus on the first. Top-down parsing can be understood as trying to find a leftmost derivation for an input string. Equivalently, top-down parsing can be seen as an attempt to construct the parse tree from the root downward in preorder.

Recursive-descent parsing is an example of top-down parsing:

- One procedure associated with each nonterminal.
- Each procedure tries to derive a prefix of the current input according to the productions for the nonterminal in question.
- Procedures for other nonterminals are invoked recursively as needed.
- Can in general involve backtracking.
- Cannot handle left-recursive grammars.

Predictive parsing is an example of recursive descent parsing where no backtracking is needed. The grammar must be such that the next input symbol uniquely determines the alternative that applies next. Many grammars can be put in the required form through grammar transformations.

A predictive recursive-descent parser is developed as follows:

- Put grammar into required form
- Eliminate left recursion
- Ensure next input symbol always determines the alternative in case of a choice (use substitution and left factoring).
- In the worst case, generalise the grammar slightly (and check the dropped restrictions after parsing).

For each symbol X in the grammar, implement a procedure parse_X . For each terminal symbol t , parse_t simply checks whether the current token is t . If so, the token is skipped.

In practice, for the terminals, implement a single procedure parameterised on the expected token.

For each nonterminal symbol A , parse_A is constructed by induction on the structure of the RHS:

- XY is implemented by $\text{parse}_X(); \text{parse}_Y();$
- $X \mid Y$ is implemented by `if (??) $\text{parse}_X();$ else if (??) $\text{parse}_Y();$ else error;`
- X^* is implemented by `while (??) $\text{parse}_X();$`

How to make the choices:

- Compute the set of terminal symbols (or starters) that can start strings derived from each alternative.
- If there is a choice between two or more alternatives, insist that the starter sets for those are disjoint.
- The right choice can now be made simply by determining to which alternative's starter set the next input symbol belongs.

Consider:

```
single-cmd  $\rightarrow$  identifier := expression  
| while expression do cmd
```

The body of the corresponding parsing procedure:

```
if (current_token is an identifier) {  
    parseIdentifier();  
    parseBecomes();  
    parseExpression();  
} else if (current_token is a while) {  
    parseWhile();  
    parseExpression();  
    parseDo();  
    parseCmd();  
} else throw an error;
```

Shift-Reduce Parsing

Shift-reduce parsing is a general style of bottom-up syntax analysis. It works from the leaves toward the root of the parse tree. There are two basic actions:

- Shift (read) next terminal symbol.
- Reduce a sequence of read terminals and previously reduced nonterminals corresponding to the RHS of a production to LHS nonterminal of that production.

A Bottom-Up Rightmost Derivation in Reverse

Consider the grammar

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow bcA \mid c \\ B &\rightarrow d \end{aligned}$$

Reduction steps for the sentence *abccde* to *S*:

abccde (reduce by $A \rightarrow c$)
abcAde (reduce by $A \rightarrow bcA$)
aAde (reduce by $B \rightarrow d$)
aABe (reduce by $S \rightarrow aABe$)
S

An *item* for a CFG is a production with a dot anywhere in the RHS.

For example, the items for the grammar

$$\begin{aligned} S &\rightarrow aAc \\ A &\rightarrow Ab \mid \epsilon \end{aligned}$$

are

$$\begin{array}{ll} S \rightarrow \bullet aAc & A \rightarrow \bullet Ab \\ S \rightarrow a \bullet Ac & A \rightarrow A \bullet b \\ S \rightarrow aA \bullet c & A \rightarrow Ab \bullet \\ S \rightarrow aAc \bullet & A \rightarrow \bullet \end{array}$$

An item is *complete* if the dot is the rightmost symbol.

A *right-sentential form* is a sentential form that can be derived by a rightmost derivation.#

A handle of a sentential form is a substring α such that α matches the RHS of a production $A \rightarrow \alpha$ and replacing α by the LHS *A* represents a step in the reverse of a rightmost derivation of *s*.

Consider the grammar

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \mid b \\ B &\rightarrow d \end{aligned}$$

For this grammar, the rightmost derivation for the input *abccde* is
 $S \rightarrow aABe \rightarrow aAde \rightarrow aAbcde \rightarrow abccde$

The string *aAbcde* can be reduced in two ways:

- $aAde \leftarrow aAbcde$ (using the first derivation of *A*)
- $aAbcBe \leftarrow aAbcde$ (using the derivation of *B*)

But the latter is not a rightmost derivation, so *Abc* is the only handle.

For an unambiguous grammar, the rightmost derivation is unique. Thus we can talk about “the handle” rather than merely “a handle”.

A *viable prefix* of a right-sentential form, *r*, is any prefix of *r* ending no further right than the right end of the handle of *r*. E.g., given the grammar

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow bcA \mid c \\ B &\rightarrow d \end{aligned}$$

the right-sentential form *abcAde* has handle *bcA*. Its viable prefixes are: ϵ , *a*, *ab*, *abc*, *abcA*.

Revision Notes

An item $A \rightarrow \beta \cdot \gamma$ represents a state of the parser where β is on the stack and the parser is looking for γ . If the parser finds γ then it might have the whole handle.

An item is *valid* for a viable prefix $\alpha\beta$ if there is a rightmost derivation $S \rightarrow^* \alpha A z \rightarrow^* \alpha\beta\gamma z$

An item is *complete* if the dot is the rightmost symbol in the item.

Example:

Given the grammar $S \rightarrow aABe$
 $A \rightarrow bcA \mid c$
 $B \rightarrow d$

the right-sentential form $abcAde$ has handle bcA . Its viable prefixes are: ϵ , a , ab , abc , $abcA$.

The item $A \rightarrow \cdot bcA$ is valid for the viable prefix a .

The item $A \rightarrow b \cdot cA$ is valid for the viable prefix ab .

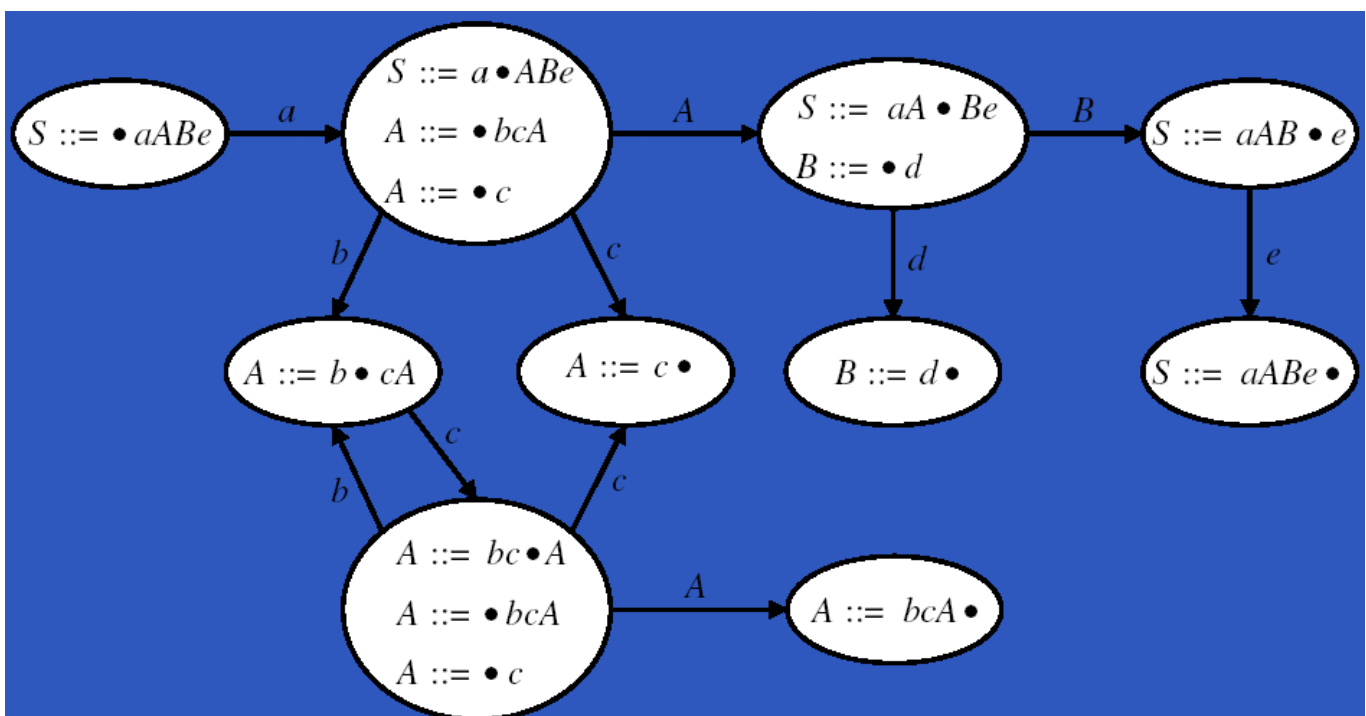
The item $A \rightarrow bc \cdot A$ is valid for the viable prefix abc .

The item $A \rightarrow bcA \cdot$ is a complete valid item for the viable prefix $abcA$.

Knowing the valid items for a viable prefix allows a rightmost derivation in reverse to be found, e.g., if $A \rightarrow \alpha \cdot$ is a complete valid item for a viable prefix γ of a string γw (where w is a sequence of terminals), then it appears that $A \rightarrow \alpha$ could have been used at the last step.

LR(0) Parsing

- A CFG for which knowing a complete valid item is enough to determine the previous right-sentential form is called LR(0).
- For every CFG whatsoever, the set of viable prefixes is regular. Thus, an efficient parser can be developed for an LR(0) CFG based on a DFA for recognising viable prefixes and their valid items.
- The states of the DFA are sets of items valid for a recognised viable prefix.



Revision Notes

Given a DFA recognising viable prefixes, a LR(0) parser can be constructed as follows:

- In a state without complete items: Shift
 - Read next terminal symbol and push it onto internal parse stack.
 - Move to new state by following edge labelled by the read terminal.
- In a state with a single complete item: Reduce
 - The top of the parse stack contains the handle of the current right-sentential form (since we have recognised a viable prefix for which a single complete item is valid).
 - The handle is just the RHS of the valid item.
 - Reduce to the previous right-sentential form by replacing the handle on the parse stack with the LHS of the valid item.
 - Move to the state indicated by the new viable prefix on the parse stack.

If a state contains both complete and incomplete items, or if a state contains more than one complete item, then the grammar was not LR(0).

LR(k) Grammars

- In practice, LR(0) tends to be a bit too restrictive.
- If we add one symbol of “lookahead” by determining the set of terminals that possibly could follow a handle being reduced by a production, then a wider class of grammars can be handled.
- Such grammars are called LR(1).

The idea is that we

- Associate a lookahead set with items: $A \rightarrow \alpha \bullet \beta, \{a_1, a_2, \dots, a_n\}$
- On reduction, a complete item is only valid if the next input symbol belongs to its lookahead set.
- In this way, it is OK to have two or more simultaneously valid complete items, as long as their lookahead sets are disjoint.

It is possible to have more than one symbol of lookahead. In general, a grammar that may be parsed with k symbols of lookahead is called LR(k). However, $k > 1$ does not add to the class of languages that can be defined.

A problem with LR(k) parsers is that the DFAs become very large. LALR (“lookahead LR”) is a simplified construction that leads to much smaller DFAs. The basic idea is to reduce the number of states by merging sets of LR(1) items that are “similar”. LALR places some additional constraints on a grammar, but those constraints are not too severe in practice. Most programming languages have LALR grammars.

Parser Generators

Constructing parsers by hand can be very tedious and time consuming. This is true in particular for LR(k) and LALR parsers: constructing the corresponding DFAs is extremely laborious. Moreover, parser construction is often a very mechanical process. So why not write a program to do the hard work for us?

A Parser Generator (or “compiler compiler”) takes a grammar as input and outputs a parser (a program) for that grammar. The input grammar is usually augmented with “semantic actions”: code fragments that get invoked when the parser successfully performs a derivation step (e.g. a reduction). The semantic actions are typically employed to construct an abstract syntax tree or to execute the program being parsed immediately (i.e., interpret it).

Some examples of parser generators include Yacc (“Yet Another Compiler Compiler”): A classic UNIX LALR parser generator for C; Bison: GNU project parser generator, a free Yacc replacement, for C and C++; Happy: a parser generator for Haskell, similar to Yacc and Bison; ANTLR: LL(k) (recursive descent) parser generator and other translator-oriented tools.

Revision Notes

Happy Parser for TXL

We are going to take a brief look at Happy. As an example, we are going to revisit TXL (the Trivial eXpression Language). See Page 2.

A simple Happy input file looks like follows:

```
{ Module Header }
%name ParserFunctionName
%tokentype TokenTypeName
%token
Specification of Terminal Symbols
```

Grammar productions with semantic actions

```
{ Further Haskell Code }
```

The terminal symbol specification specifies terminals to be used in productions and relates them to Haskell constructors for the tokens:

```
%token
    int      { T_Int $$ }
    ident    { T_Id  $$ }
    '+'      { T_Plus }
    '-'      { T_Minus }
    ...
    '='      { T_Equal }
    let      { T_Let  }
    in       { T_In   }
```

The grammar productions are more or less written in BNF, with an additional semantic action defining the return value for reduction by the production in question:

```
add_exp      : mul_exp          { $1 }
              | add_exp '+' mul_exp { BinOpApp Plus $1 $3 }
              | add_exp '-' mul_exp { BinOpApp Minus $1 $3 }
```

See the full example HappyTXL.y for further details.

Precedence and Associativity

Happy (like e.g. Yacc and Bison) allows operator precedence and associativity to be explicitly specified to disambiguate a grammar:

```
%left '+' '-'
%left '*' '/'
exp      : exp '+' exp { BinOpApp Plus $1 $3 }
          | exp '-' exp { BinOpApp Minus $1 $3 }
          | exp '*' exp { BinOpApp Times $1 $3 }
          | exp '/' exp { BinOpApp Divide $1 $3 }
```

See HappyTXL2.y for further details.

A TXL Interpreter

The semantic actions do not have to construct an AST. An alternative (for simple languages, at least) is to execute the code being parsed immediately. I.e. to construct an interpreter. The example HappyTXLInterpreter.y shows one way. Each semantic action returns a function of type `Env -> Int`
`Type Env = Id -> Int` (i.e., maps an identifier to a value)

The semantic action for a variable looks up the variable value in the environment. The semantic action for `let` extends the argument environment and evaluates the body in the extended environment.

See HappyTXLInterpreter.y for further details.

Contextual Analysis

The Visitor Pattern

Consider a simple tree data type in Haskell, and a function to count the nodes in a tree:

```
data Tree = Leaf Int
          | Node2 Tree Tree
```

```
count :: Tree -> Int
count (Leaf _) = 1
count (Node2 t1 t2) = 1 + count t1 + count t2
```

Suppose we want to write a function that triples the values in the leaf nodes:

```
triple :: Tree -> Tree
triple (Leaf n) = Leaf (3 * n)
triple (Node2 t1 t2) = Node2 (triple t1) (triple t2)
```

Everything else remains unchanged. Adding new functionality is a strictly local change.

Now suppose we want to add a new kind of tree node with three subtrees. We then have to:

- extend the data type definition with new constructor
- extend count with one new case
- extend triple with one new case.

In general, changes have to be made wherever the structure of values of the changed type is analysed.

Languages where functions/procedures and data types are defined separately (like Haskell, C) make:

- adding new functionality easy, since changes are local
- changing data types hard, since changes are global.

Let us redo the example in standard OO style in Java, i.e. with functions and data defined together:

```
public abstract class Tree {
    public abstract int count();
}

public class Leaf extends Tree {
    protected int n;
    public Leaf(int n) {
        this.n = n;
    }
    public int count() {
        return 1;
    }
}

public class Node2 extends Tree {
    protected Tree t1, t2;
    public Node2(Tree t1, Tree t2) {
        this.t1 = t1;
        this.t2 = t2;
    }
    public int count() {
        return 1 + t1.count() + t2.count();
    }
}
```

Revision Notes

In OO style, adding a new kind of tree node is easy. Just add a new class (local change). On the other hand, adding new functionality (i.e. new methods) like triple is hard: all involved classes must be changed (non-local changes).

Languages where functions/procedures and data types are defined together (like Java, C++, C#) make:

- adding new functionality hard, since changes are global;
- changing data types easy, since changes are local.

The situation is exactly the opposite of the one in languages like Haskell.

Suppose we are using an OO language, but that we still would like to make it easy to add new functionality. Is that possible? I.e., can the definition of data be separated from the definition of functionality in an OO language? The answer, of course, is yes and this is what the *Visitor Design Pattern* achieves.

The Visitor Pattern: Key Ideas

- Code for implementing some particular piece of functionality is collected in a visitor class.
- A visitor class has one instance method for each non-abstract class that defines the data structure on which the visitor class is intended to operate.
- Visit methods defined in the classes of the data structure effectively implement a case statement by invoking the right instance method of the visitor object supplied as argument.

The Visitor Interface

All visitor classes intended to operate on a particular data structure need to share a common *interface*. In our case:

```
public interface Visitor {  
    public Object visitLeaf(Leaf leaf, Object varg);  
    public Object visitNode2(Node2 node2, Object varg);  
    public Object visitNode3(Node3 node3, Object varg);  
}
```

- One method for each non-abstract class defining the data structure in question.
- Each method takes the node on which it is invoked as its first argument.
- Each method has an extra argument in case that is needed for some visitor classes.
- Each method returns a result in case that is needed for some visitor classes.
- The type of the extra argument/result is Object to make it as general as possible.
- Specific visitor classes are likely to operate on more specific types than Object.
- Thus type casts are frequently needed in the actual implementations of the methods.
- Note that the visitor interface is specific to a particular data structure.

Each data structure class defines a *visit method*:

```
public abstract class Tree {  
    public abstract Object visit(Visitor v, Object varg);  
}
```

The method takes a *visitor object*, a visitor class instance, as its first argument. This object “carries” the code to be invoked.

The method has an *extra argument and result* like the visitor class methods.



The non-abstract class Leaf

```
public class Leaf extends Tree {
    protected int n;
    public Leaf(int n) { this.n = n; }
    public Object visit(Visitor v, Object varg){
        return v.visitLeaf(this, varg);
    }
}
```

- Invokes the visitor method for leaves.
- The type of this is Leaf here.
- Result and extra argument propagated.
- Note how the visit methods together effectively implements a case over the tree type through dynamic dispatch.
- A distinct visitor method gets invoked for each possible type of tree node.

Example Visitor Class: Count (A visitor class for counting nodes in a tree)

```
public final class Count implements Visitor {
    public Object visitLeaf(Leaf leaf, Object varg) {
        return new Integer(1);
    }
}
```

- Each visitor class must *implement the Visitor interface*.
- The *extra argument* is not used in this case.
- The returned value must be of *reference type*.

```
Object visitNode2(Node2 node2, Object varg) {
    return new Integer(1) + ((Integer) node2.t1.visit(this, null)).intValue()
        + ((Integer) node2.t2.visit(this, null)).intValue();
}
```

- The visit method gets invoked recursively on each subtree with the *current visitor object as the first argument*.

Using Visitor Classes

Calling a visitor is a two stage process:

- First *instantiate the visitor class*: private static Count count = new Count();
- Then *invoke* the visit method on the data structure to be traversed with the *visitor as argument*.
Integer n = (Integer) aTree.visit(count, null);

Contextual Analysis

Among other things, this involves:

- resolving meaning of symbols
- reporting undefined symbols
- type checking

In short, contextual analysis is about ensuring that a program is *statically well-formed*.

But syntax has to do with “form”. So what is new? Can’t we use context-free grammars to express e.g. type constraints and thus make the parser do the checking for us?

Limitations of CFGs

- If we try and express a “declare before use” requirement using a CFG, it soon becomes obvious that things get quite complicated.
- In fact, the number of nonterminals (and thus the number of productions) grow exponentially, i.e., for n variables, 2^n expr-vars nonterminals are needed.
- Normally, the number of variables is *unlimited*. That would imply *infinitely* many productions. But that would no longer be a CFG!

If we try and include type-checking into the grammar, although it might look reasonable at first sight:

- The whole scheme hinges on having partitioned the variables into two groups: *integer variables* (int-var) and *boolean variables* (bool-var).
- But in most languages the *name* of a variable is independent of its type. In fact, distinct variables with the same name could be used at different types in a single program. Similarly, if we tried to generalise to incorporate user-defined functions, those would also have to be partitioned into groups of different types based on their names. Not a realistic assumption!

Our examples do not prove that it is impossible to achieve what we tried to achieve using CFGs. However, it *can* be proved that this indeed is the case: contextual constraints result in *context sensitive* languages; such languages *cannot* be described by CFGs.

It can be shown that unrestricted grammars are equivalent to Turing Machines. Thus, to check contextual constraints, we need to write programs in a Turing-equivalent language. Simpler machines such as PDAs (Push-Down Automata) are not powerful enough.

Contextual Analysis is about checking contextual constraints. Typically two kinds of constraints - scope rules (visibility, which declarations take effect where) and type rules (internal consistency, ensuring that every expression computes a value of acceptable form, i.e., has a valid type). Contextual analysis thus typically consist of two subphases (at least conceptually):

- Identification: applying the scope rules in order to relate each applied identifier occurrence to its declaration.
- Type checking: applying the type rules to infer the type of each expression and compare it with the expected type.

Scope rules and type rules are not the only possible kinds of contextual constraints. For example, Java has rules concerning abstract and final classes (e.g., only abstract classes can have abstract methods, abstract classes may not be instantiated, a final class cannot be extended, a class cannot be both final and abstract, etc.), exceptions (e.g., the set of exceptions a method can raise must be declared (except for unchecked exceptions)) and definite assignment (a local variable must not be read unless it has been “definitely assigned before”).



Identification

Identification is the task of relating each applied identifier occurrence to its declaration.

```
public class C {
    int x, n;
    void set(int n) { x = n; }
}
```

In the body of the method set, the one applied occurrence of x refers to the instance variable x whereas n refers to the argument n.

Scope and Scope Rules

The identification process is governed by the scope rules of the language:

- scope: the portion over a program over which a declaration takes effect.
- block: a program phrase that delimits the scope of declarations within it.

Consider the MiniTriangle let block command, let decls in body. The scope of each declaration is the rest of the block. For example,
let

	const m ~ 10;		
	const n ~ m * 2		scope of m
in	putint(n);		scope of n

In addition to deciding the range of declarations, the scope rules also deal with issues like whether explicit declarations are required, whether multiple declarations at the same level are allowed or whether shadowing/hiding is allowed.

Some Java Scope Rules

- The scope of a type introduced by a class type declaration or interface type declaration is the declarations of all class and interface types in all the compilation units of the package in which it is declared.
- The scope of a member declared in or inherited by a class type or interface type is the entire declaration of the class or interface type. The declaration of a member needs to appear before it is used only when the use is in a field initialization expression.
- The scope of a parameter of a method is the entire body of the method.
- The scope of a local variable declaration in a block is the rest of the block in which the declaration appears, starting with its own initializer and including any further declarators to the right in the local variable declaration statement.
- Hiding the name of a local variable is not permitted. For example, the following code fragment is rejected:

```
static int x = 10;
public void foo(int x) {
    if (x < 0) {
        int x = 10;
        ...
    }
    ...
}
```

It is OK to hide the class variable, but not OK to hide the method parameter.



Identification/Symbol Table

An identification table, also called symbol table or environment, is typically used during identification to keep track of symbols and their attributes, such as:

- kind of symbol (class name, local variable, etc.)
- scope level
- type
- source code position

The organisation of the symbol table depends on the source language's block structure. Three main possibilities:

- Monolithic block structure: one common, global scope (Old Basic dialects, Cobol, Assembly lang., etc.)
- Flat block structure: blocks with local scope enclosed in a global scope. (Fortran)
- Nested block structure: blocks can be nested to arbitrary depth. (Ada, C, C++, Java, C#, Haskell...)

We will focus on nested block structure in the following since monolithic and flat block structure can be considered special cases of nested block structure and variations on nested block structure is by far the most common in modern high-level languages.

Using the Symbol Table

The symbol table is used as follows during identification:

- Initialise the table, e.g. enter the standard environment.
- When a declaration is encountered:
 - check if declared identifier clashes with existing symbol and report error if it does.
 - If not, enter declared identifier into table along with its attributes.
- When an applied identifier occurrence is encountered:
 - look up identifier in table, taking scope rules into account and report an error if not found
 - if found, annotate applied occurrence with symbol attributes from table.
- When entering a new block, arrange so that subsequently entered symbols become associated with the scope corresponding to the block ("open scope").
- When leaving a block, remove/make inaccessible symbols declared in that block ("close scope").

The MiniTriangle Symbol Table

The MiniTriangle symbol table is similar to what was used in the LTXL example.

The table is represented by a linked list:

```
public class IdEntry {
    protected String id;
    protected Declaration attr;
    protected int level;
    protected IdEntry previous;
    IdEntry (...) { ...}
}
```

However, there is only one copy of the table. It is updated imperatively (or destructively). The imperative style necessitates "open scope" and "close scope" operations. The close scope operation restores (imperatively) the table to what it was before the previous open scope operation. In the pure functional setting, references to as many versions of the symbol table as were needed were kept. No need for a close scope operation.

Revision Notes

```
public final class IdentificationTable {
    private int level;
    private IdEntry latest;

    public IdentificationTable () {...}

    public Declaration retrieve (String id) {...}

    public void openScope () { level++; }

    public void closeScope () {
        IdEntry entry, local;
        entry = this.latest;
        while (entry.level == this.level) {
            local = entry;
            entry = local.previous;
        }
        this.level--;
        this.latest = entry;
    }

    public boolean declaredAtThisLevel (String id) {
        IdEntry entry = this.latest;
        while ((entry.level == this.level) && !(entry.id.equals(id))) entry = entry.previous;
        return (entry.level == this.level);
    }

    public void enter (String id, Declaration attr) {
        IdEntry entry;
        if ((this.level == 0) || !this.declaredAtThisLevel(id)) {
            entry = new IdEntry(id, attr, this.level, this.latest);
            this.latest = entry;
        }
    }

    public Declaration retrieve (String id) {
        IdEntry entry = this.latest;
        Declaration attr = null;
        boolean present = false, searching = true;

        while (searching) {
            if (entry == null) searching = false;
            else if (entry.id.equals(id)) {
                present = true;
                searching = false;
                attr = entry.attr;
            }
            else entry = entry.previous;
        }
        return attr;
    }
}
```



Revision Notes

Lists don't make very efficient symbol tables. Insertion (at head) is fast, $O(1)$, but lookup is $O(n)$, where n is the number of symbols. Some more efficient options:

- Balanced trees:
 - Insertion and lookup are both $O(\log n)$
 - One way of handling nested scopes would be a stack of trees.
- Hash tables:
 - Insertion and lookup are both $O(1)$ as long as the ratio between the number of symbols and the hash table size is kept below a small constant factor.
 - Algorithms such as linear hashing allows the table to grow and shrink gracefully, guaranteeing near optimal performance.

Types and Type Systems

Type systems are an example of lightweight formal methods (highly automated but with limited expressive power). A type system is "a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

- Static checking is implied since goal is to prove absence of certain errors.
- "Dynamically typed" languages do not have a type system according to this definition: they should really be called dynamically checked.
- Emphasis on classification of syntactic phrases or terms: a type system computes a static approximation of the run-time behaviour.
- A type system is necessarily conservative: some programs that actually behave well will be rejected.
- A type system checks for certain kinds of program behaviour or run-time errors. Exactly which depends on the type system and the language design.
- The safety or soundness of a type system must be judged with respect to its own set of run-time errors.

Language Safety

Language safety is a contentious notion. A safe language is "one that protects its abstractions".

- Language safety is not the same as static type safety: language safety can be achieved through static type checking and/or dynamic run-time checks.
- Scheme is an example of a dynamically checked safe language.
- Even languages with static type checking usually use some dynamic checks to achieve language safety, e.g.:
 - checking of array bounds
 - down casting (e.g. java)

Some examples of statically and dynamically checked safe and unsafe languages:

	Statically chkd	Dynamically chkd
Safe	ML, Haskell, Java	Lisp, Scheme, Perl, Python, Postscript
Unsafe	C, C++	Certain Basic dialects

An Example Language

Terms:

- | | | |
|---|--------------------|----------------|
| t | → true | constant true |
| | false | constant false |
| | if t then t else t | conditional |
| | 0 | constant zero |
| | succ t | successor |
| | pred t | predecessor |
| | iszero t | zero test |

Revision Notes

The values of a language are a subset of the terms that are possible results of evaluation. The evaluation rules are going to be such that no evaluation is possible for values. A term to which no evaluation rule applies is a normal form. All values are normal forms.

Values:

$v \rightarrow \text{true}$	true value
false	false value
nv	numeric value

Numeric Values:

$nv \rightarrow 0$	zero value
$\text{succ } nv$	successor value

The One Step Evaluation Relation

$t \rightarrow t'$ is an evaluation relation on terms. Read “ t evaluates to t' in one step”. The evaluation relation constitute an operational semantics for the example language.

$\text{if true then } t_2 \text{ else } t_3 \rightarrow t_2$	(E-IFTRUE)
$\text{if false then } t_2 \text{ else } t_3 \rightarrow t_3$	(E-IFFALSE)

$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$	(E-IF)
---	--------

$\frac{t_1 \rightarrow t_1'}{\text{succ } t_1 \rightarrow \text{succ } t_1'}$	(E-SUCC)
---	----------

$\text{pred } 0 \rightarrow 0$	(E-PREDZERO)
$\text{pred } (\text{succ } nv_1) \rightarrow nv_1$	(E-PREDSUCC)

$\frac{t_1 \rightarrow t_1'}{\text{pred } t_1 \rightarrow \text{pred } t_1'}$	(E-PRED)
---	----------

$\text{iszero } 0 \rightarrow \text{true}$	(E-ISZEROZERO)
--	----------------

$\text{iszero } (\text{succ } nv') \rightarrow \text{false}$	(E-ISZEROSUCC)
--	----------------

$\frac{t_1 \rightarrow t_1'}{\text{iszero } t_1 \rightarrow \text{iszero } t_1'}$	(E-ISZERO)
---	------------

Note that:

- Values cannot be evaluated further. E.g.: true, 0, succ (succ 0)
- Certain “obviously nonsensical” states are stuck: the term cannot be evaluated further but it is not a value. For example: if 0 then pred 0 else 0
- We let the notion of getting stuck model run-time errors.
- The goal of a type system is thus to guarantee that a program never gets stuck.



Revision Notes

At this point, there are only two types, booleans and the natural numbers:

Types:

$T \rightarrow \text{Bool}$	type of booleans
$ \text{Nat}$	type of natural numbers

Typing Rules:

$\text{true} : \text{Bool}$	(T-TRUE)
$\text{false} : \text{Bool}$	(T-FALSE)

$\frac{t1 : \text{Bool} \quad t2 : T \quad t3 : T}{\text{if } t1 \text{ then } t2 \text{ else } t3 : T}$	(T-IF)
--	--------

$0 : \text{Nat}$	(T-ZERO)
------------------	----------

$\frac{t1 : \text{Nat}}{\text{succ } t1 : \text{Nat}}$	(T-SUCC)
--	----------

$\frac{t1 : \text{Nat}}{\text{pred } t1 : \text{Nat}}$	(T-PRED)
--	----------

$\frac{t1 : \text{Nat}}{\text{iszero } t1 : \text{Bool}}$	(T-ISZERO)
---	------------

Safety = Progress + Preservation

The most basic property of any type system is safety, or “well typed programs do not go wrong”, where “wrong” means entering a “stuck state”. This breaks down into two parts:

- Progress: A well-typed term is not stuck.
- Preservation: If a well-typed term takes a step of evaluation, then the resulting term is also well-typed.
- Together, these properties say that a well-typed term can never reach a stuck state during evaluation.

Formally:

- THEOREM [PROGRESS]: Suppose that t is a well-typed term (i.e., $t : T$), then either t is a value or else there is some t_0 with $t \rightarrow t_0$.
- PROOF: By induction on a derivation of $t : T$.
- THEOREM [PRESERVATION]: If $t : T$ and $t \rightarrow t_0$ then $t_0 : T$.
- PROOF: By induction on a derivation of $t : T$.

Progress: A Proof Fragment

A typical case when proving progress by induction on a derivation of $t : T$.

Case T-IF:

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$
where $t_1 : \text{Bool}, t_2 : T, t_3 : T$

By ind. hyp, either t_1 is a value, or else there is some t'_1 such that $t_1 \rightarrow t'_1$.

If t is a value, then it must be either true or false, in which case either E-IFTRUE and E-IFFALSE applies to t . On the other hand, if $t_1 \rightarrow t'_1$, then by T-IF, $t \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$.



Revision Notes

Let Bindings

Syntactic extension:

$t \rightarrow x$	variable
let $x = t$ in t	let binding

New evaluation rules:

let $x = v_1$ in $t_2 \rightarrow [x \mid - v_1] t_2$ (E-LETV)

$$\frac{t_1 \rightarrow t'_1}{\text{let } x = t_1 \text{ in } t_2 \rightarrow \text{let } x = t'_1 \text{ in } t_2} \quad (\text{E-LET})$$

We now need a typing context or type environment to keep track of types of variables (an abstract version of the “identification table”). The typing relation thus become a trinary relation $\Gamma \mid - t : T$, read “term t has type T in typing context Γ ”.

Updated typing rules:

$\Gamma \mid - \text{true} : \text{Bool}$	(T-TRUE)
$\Gamma \mid - \text{false} : \text{Bool}$	(T-FALSE)

$$\frac{\Gamma \mid - t_1 : \text{Bool} \quad \Gamma \mid - t_2 : T \quad \Gamma \mid - t_3 : T}{\Gamma \mid - \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (\text{T-IF})$$

etc.

New typing rules:

$$\frac{x : T \in \Gamma}{\Gamma \mid - x : T} \quad (\text{T-VAR})$$

$$\frac{\Gamma \mid - t_1 : T_1 \quad \Gamma, x : T_1 \mid - t_2 : T_2}{\Gamma \mid - \text{let } x = t_1 \text{ in } t_2 : T_2} \quad (\text{T-LET})$$

Functions

Syntactic extension:

$t \rightarrow \lambda x : T. t$	abstraction
$t t$	application

Values:

$v \rightarrow \lambda x : T. t$	abstraction value
----------------------------------	-------------------

Types:

$T \rightarrow T \rightarrow T$	type of functions
---------------------------------	-------------------

New evaluation rules:

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2} \quad (\text{E-APP2})$$

$$(\lambda x : T_{11}. t_{12}) v_2 \rightarrow [x \mid - v_2] t_{12} \quad (\text{E-APPABS})$$

Note:

- left to right evaluation order: first the function (E-APP1), then the argument (E-APP2)
- call-by-value: the argument fully evaluated before function “invoked” (E-APPABS).



New typing rules:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad (\text{T-APP})$$

Some proof derivation examples:

Prove if (iszero (pred (succ 0))) then (pred 0) else (succ 0) is well-formed.

1.

$$\frac{\frac{\text{pred (succ 0)} \rightarrow 0}{\text{iszero (pred (succ 0))} \rightarrow \text{iszero(0)}}}{\text{if (iszero (pred (succ 0))) then (pred 0) else (succ 0)} \rightarrow \text{if (iszero (0)) then (pred 0) else (succ 0)}} \quad \begin{array}{l} \text{E-PREDSUCC} \\ \text{E-ISZERO} \\ \text{E-IF} \end{array}$$

2.

$$\frac{\text{iszero(0)} \rightarrow \text{true}}{\text{if (iszero 0) then (pred 0) else (succ 0)} \rightarrow \text{if (true) then (pred 0) else (succ 0)}} \quad \begin{array}{l} \text{E-ISZEROZERO} \\ \text{E-IF} \end{array}$$

3.

$$\text{if (true) then (pred 0) else (succ 0)} \rightarrow \text{true} \quad \text{E-IFTRUE}$$

4.

$$\text{(pred 0)} \rightarrow 0 \quad \text{E-PREDZEROS}$$

Now check the types:

$$\frac{\frac{\frac{\text{0:Nat}}{\text{succ 0:Nat}}}{\text{pred (succ 0):Nat}} \quad \frac{\text{0:Nat}}{\text{pred 0:Nat}} \quad \frac{\text{0:Nat}}{\text{succ 0:Nat}}}{\text{if (iszero (pred (succ 0))):Nat then (pred 0):Nat else (succ 0):Nat}} \quad \text{E-IF}$$

Run-Time Organization

Key issues:

- Data representation: How to represent each source language type?
- Storage allocation: Where to store values of variables, and how to manage allocation and deallocation, reflecting their life time?
- Expression evaluation: How to take care of intermediate results?
- Routines: How to pass parameters, arrange local storage, access all variables in scope?

Data Representation

Nonconfusion

Different values of a given type must have different representations.

This is self-evident - if two different values are represented the same way, they cannot be told apart. In an untyped language, every possible value must have a distinct representation but in a (statically) typed language, values of the same type must have distinct representations - the same representation may be reused for values of different types.

Problem: A digital computer with a finite memory cannot represent every possible real number exactly. (In fact, cannot represent every possible integer either.)

Solution: Choose to represent a subset of the real numbers; represent each real with the closest member in the representable set.

Result: Computations that “mathematically” ought to yield different values could in fact give the same result.

However, computers don't know anything about “real” numbers. There is only a problem if we forget/pretend this not to be the case. Floating-point representation is usually chosen. The rules for floating-point arithmetic are precisely defined and each floating-point value is distinct from every other floating-point value so there is no confusion as long as understood that operating with floating-point numbers.

Situation is similar for integers. Consider unsigned addition on a 16-bit computer:
 $65535 + 1 = 1111111111111112 + 1 = 0$,
or possibly an overflow error.

The result is not what one would expect mathematically when computing with integers. We have to remember that the semantics of the underlying operations is different from the standard arithmetic operations.

Uniqueness

Each value should have exactly one representation.

The comparison of values is facilitated if each value has exactly one representation, although it is not essential. Common exceptions:

In ones-complement, 0 is represented by both 00...00 and 11...11

Floating-point representations typically have a separate sign bit. Thus, the representation of +0 is distinct from the representation of -0

Constant-size representation

The representations of all values of a given type should occupy the same amount of space. Constant-size representation enables the compiler to statically plan storage allocation (since type and hence size is known statically).



Direct or indirect (via pointer) representation

- Direct representation: the representation of a value x is the representation of x
 - efficient access
 - no heap allocation/deallocation overhead
- Indirect representation: x represented by a handle that points to the representation of x (on the stack or in the heap)
 - supports varying size data (like dynamic arrays)
 - supports recursive types (like linked lists, trees)
 - facilitates implementation of polymorphism

Representing Primitive Types

Primitive types are often supported directly by the underlying hardware, for example, a 32-bit machine might support

- addressing of 8-bit bytes and 32-bit words
- 32-bit twos-complement integer arithmetic
- 64-bit floating point operations

There may also be standard encoding conventions, such as the 7-bit ASCII or 8-bit ISO character codes.

On such a 32-bit machine, the following would be natural representation choices:

Type	Representation	Size
Boolean	0 for false; 1 for true	8-bit byte
Char	ISO Latin 1 encoding	8-bit byte
Integer	twos-complement repr.	32-bit word
Real	floating point repr.	64-bit word

A record consists of several fields, each of which has an identifier. For example:

```

type Date = record
    d: Integer,
    m: Integer,
    y: Integer
end;
type Details = record
    female: Boolean,
    dob: Date,
    status: Char
end;

```

A record is a sequence of representations of individual fields. Beware of alignment restrictions such as the underlying architecture requiring that e.g. word-sized quantities start at a word boundary.

Example: Assume a variable x of type Details (as above) with the requirement that Integers be word-aligned. This means:

- variables of type Date must be word-aligned since it has Integer fields
- variables of type Detail must be word-aligned since has a word-aligned field Date
- the field female must be padded to word length to ensure that the field dob is word-aligned.

Arrays are represented by a sequence of representations of the individual array elements. The address of individual array element can be easily computed from the base address and index since the size of the individual elements is known, e.g., $a = \text{addr}(\text{cs}[0]) + i \times \text{size}(\text{Char})$.

With static arrays, the size is known at compile time. The required storage space is known statically and so the index can be checked as being within bounds by comparing with statically known constants.

Revision Notes

With dynamic arrays, the size not known at compile time. There is an indirect representation with the array being accessed via a handle, which contains a pointer to actual array along with array bounds. The storage for the array is allocated at runtime and the index can be checked to be within bounds by comparing with array bounds stored in the handle.

Representing Disjoint Unions

A disjoint union consists of a tag and a variant part, in which the value of the tag determines the type of the variant part. Disjoint unions occur as variant records in Pascal and Ada and as data types in Haskell and ML. They can be represented like a record where the tag is a field whose value determines the layout of the rest of the record. If constant size is necessary, we take the size to be the maximal size over the various possible layouts.

Some Haskell Examples:

```
data OptInt = NoInt | JustInt Int
```

The first tag is NoInt. In this case, there is no variant part, which is the same as saying that we have a trivial variant part of the unit type (). The second tag is JustInt. In this case, the variant part is a single integer field.

```
data Colors = Red | Green | Blue
```

the variant part is always the unit type hence this is thus just an enumeration type.

Representing Recursive Types

A recursive type is one defined in terms of itself such as linked lists and trees. Recursive types are usually represented indirectly since this allows values of arbitrary size to be referenced through a fixed size handle.

In languages like Pascal, the programmer needs to introduce the indirect representation explicitly through pointer types:

```
type IntList = ^IntNode;
type IntNode = record
    head: Integer;
    tail: IntList
end;
var primes: IntList
```

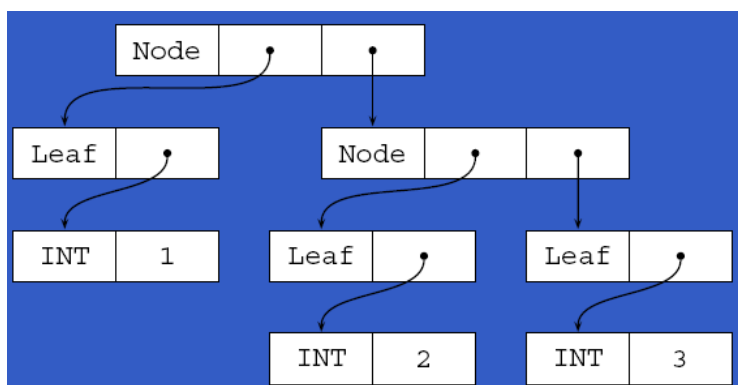
Languages like Haskell and ML adopts a uniform data representation: all values have an indirect representation (a pointer), even values of “primitive” types. This uniform representation enables parametric polymorphism:

For example, the identity function $id\ x = x$ works for values of any type since all values are represented in exactly the same way (by a pointer). The uniform representation also means that recursive types can be supported automatically: “everything is already a pointer”.

Example: Haskell Tree Type

```
data Tree = Leaf Int
         | Node Tree Tree
```

```
aTree = Node 1 (Node 2 3)
```



Of course, the tags (Leaf, Node, and INT) must also be represented. Two possibilities:



Revision Notes

- A small integer, subject to nonconfusion, e.g. Leaf = 0; Node = 1; INT = 0 (Representing both Leaf and INT with the small integer 0 does not lead to confusion in a statically typed language like Haskell.)
- A pointer to an information table.

An important issue for languages with automatic storage deallocation or garbage collection is how to distinguish between pointers (addresses) and non-pointers (e.g. tags, integers, etc.). Pointers and non-pointers are represented differently; e.g., a few reserved bits of each word encode what kind of data the word represents. The tag is associated with information that describes the layout of the object; e.g., in an information table.

Storage Allocation

Global variables exist throughout the program's run-time. Storage for such variables can thus be allocated statically, at compile time, once and for all.

Parameters and local variables exist only during a function/procedure invocation. Function/procedure calls are properly nested. In case of recursion, a function/procedure may be re-entered any number of times. Each function/procedure activation needs a private set of parameters and local variables. These observations suggest that storage for parameters and local variables should be allocated on a stack.

Stack frames

One stack frame for each currently active function/procedure. Each frame contains

- space for local parameters and local variables
- return address (location of instruction immediately following calling instruction)
- link to restore stack after exit (dynamic link)
- link to enable access to non-local variables (static link)
- contents of any registers that must be restored on exit
- temporary workspace, e.g. for expression evaluation

The Triangle Abstract Machine (TAM) has a number of registers related to the stack. Among others:

- SB: Stack Base
- ST: Stack Top
- LB: Local Base

Dynamic and Static Links

- Dynamic Link: Value to which LB (Local Base) is restored after exiting procedure. "Dynamic" because it depends on where procedure was called from.
- Static Link: Base of underlying frame of routine that immediately *lexically* encloses this one. "Static" because it depends on the program's structure and not on its execution. Used to determine addresses of non-local variables.

Consider nested procedures:

```

proc P
  var x, y, z: Integer
  proc Q
    ...
  begin ...end
proc R
  ...
  begin ...Q() ...end
begin ...R() ...end

```

P's variables are in scope also in Q and R. But how to access them? They are neither global, nor local!



Revision Notes

In particular, we cannot access x , y , z relative to the stack base (SB) since we cannot (in general) statically know if P was called directly from the main program or indirectly via one or more other procedures, i.e., there could be arbitrarily many stack frames *below* P 's frame. We cannot access x , y , z relative to the local base (LB) since we cannot (in general) statically know if e.g. Q was called directly from P , or indirectly via R or recursively via itself, i.e., there could be arbitrarily many stack frames *between* Q 's and P 's frame.

Solution:

- the Static Links in Q 's and R 's frames are set to point to P 's frame on each activation.
- the static link in P 's frame is set to point to the frame of its closest enclosing procedure and so on.

Thus, by following the chain of static links, one can access variables at any level of a nested scope. The static link is related to the static, *textual* nesting of procedures/functions, whereas the dynamic link is related to the dynamic nesting established by the caller/callee relationship. The static link refers to the activation record of the **textually** enclosing procedure/function. The dynamic link refers to the activation record of the procedure/function from which the currently active procedure/function was called.

Code Generation

Given the run-time organization, the code generator must address the following issues:

- *Code Selection*: Which code sequence to generate for each source code phrase?
- *Storage Allocation*: Which storage address to assign to each variable?
 - Global variables: static allocation
 - Local variables: relative to stack frame
- *Register Allocation*: How to allocate registers for variables etc.?

Register Allocation

The number of registers are rarely more than a few dozen. Usually not enough for all variables, Arguments and intermediate results! However, not all variables are in use all the time.

Issues:

- Deciding which register to use for which variable(s). This is *Register Allocation*.
 - If possible, don't use more registers than available!
 - *Graph Colouring* is a common method for allocating registers.
- What if too many variables are in use simultaneously?
 - *Register Pressure*: the number of registers needed for a code fragment.
 - *Register Spilling*: storing the content of a register into memory so as to free it and thus reduce the register pressure.

Graph Colouring

Basic idea of register allocation using graph colouring:

- Represent each variable by a node in a graph.
- Add an edge between two nodes if the variables are in use simultaneously.
- Colour the graph so that no two *adjacent* nodes get the same colour, using as few colours as possible.
- Each colour corresponds to a register.

Complications

In practice, register allocation can be very involved and highly architecture-specific:

- Lots of registers vs. very few registers.
- General purpose registers vs. special purpose registers, e.g. dedicated registers for result of multiplication, array indices, etc.
- register allocation and code selection highly interdependent
- Uniform size vs. varying size.
- Optimization: e.g. how to choose register to spill?



Revision Notes

If register allocation is so complicated, why bother? Why not simply keep everything in the main memory at all times? Performance! Reading/writing main memory is *orders of magnitude* slower than reading/writing registers.

To keep things simple, the Triangle Abstract Machine (TAM) is a *stack machine*:

- Only a few dedicated registers, e.g. SB: Stack Base, ST: Stack Top, LB: Local Base (or Frame Pointer)
- Variables always on the stack.
- Arguments passed via the stack.
- Temporary storage (e.g. for expression evaluation) on the stack.

Thus, for TAM, register allocation is a non-issue! But one should not expect high performance.

Specifying Code Selection

Code selection is specified *inductively* over the phrases of the source language:

```
Command    →  Identifier := Expression
           |  Command ; Command
...

```

The *Code Function* maps a source phrase to an instruction sequence. For example:

```
execute : Command → Instruction*
evaluate : Expression → Instruction*
```

Code functions are specified by means of *code templates*:

```
execute [C1; C2] =  execute C1
                  execute C2
```

The template brackets ([and]) enclose pieces of *concrete syntax* and *meta variables*. Note the *recursion*; i.e. inductive definition over the underlying phrase structure.

In a simple language, the code template for assignment might be:

```
execute [I := E] =  evaluate E
                  STORE addr (I)
```

Note that the instruction sequences and individual instructions in the RHS of the defining equation are implicitly *concatenated*. Now consider generating code for the fragment:

```
f := f * n;
n := n - 1
```

```
execute [f := f*n; n := n-1] =  execute [ f := f*n ]
                              execute [ n := n-1 ]
                              =  evaluate [ f*n ]
                              STORE addr ([f])
                              evaluate [n-1]
                              STORE addr ([n])
                              =  ...
                              =  LOAD addr([f])
                              LOAD addr([n])
                              CALL mult
                              STORE addr([f])
                              LOAD addr([n])
                              CALL pred
                              STORE addr([n])
```



Code Functions for Mini-Triangle:

run : Program \rightarrow Instruction*
 execute : Command \rightarrow Instruction*
 evaluate : Expression \rightarrow Instruction*
 fetch : Identifier \rightarrow Instruction*
 assign : Identifier \rightarrow Instruction*
 elaborate : Declaration \rightarrow Instruction*

The code function elaborate is clearly responsible for *assigning* addresses to variables. Equally clear is that fetch and assign need *access* to the addresses assigned by elaborate, but the given type signatures for the code functions do not *permit* this communication!

The code functions ought to have an extra *environment argument*, associating variables with addresses. Additionally, the code function elaborate ought to return an updated environment.

The actual implementation relies on *side effects*. During elaboration, the AST is *updated* by attaching an *entity descriptor* to the elaborated declaration. As a result of identification, all identifiers already refer to the corresponding declaration through pointers. When the address of a variable is needed, this can be obtained by dereferencing the pointer to the declaration in question.

Let us declare the following meta variable conventions:

C – Command
 E – Expression
 D – Declaration
 I – Identifier
 O – Operator
 IL – IntegerLiteral

run : Program \rightarrow Instruction*

execute : Command \rightarrow Instruction*

run [C] = execute C
 HALT

execute [I := E] = evaluate E
 assign I

execute [I(E)] = evaluate E
 CALL addr (I)

execute [C1; C2] = execute C1
 execute C2

execute [if E then C1 else C2] = evaluate E
 JUMPIF(0) g
 execute C1
 JUMP h
 g : execute C2
 h :

execute [while E do C] = JUMP h
 g : execute C
 h : evaluate E
 JUMPIF(1) g

execute [let D in C] = elaborate D
 execute C
 POP(0) s (where s is the amount of storage allocated by D)

evaluate : Expression \rightarrow Instruction*

(Note: all operations take arguments from the stack and writes result back onto the stack)

evaluate [I] = fetch I



Revision Notes

evaluate [O E] = evaluate E
 CALL addr (O)
evaluate [E1 O E2] = evaluate E1
 evaluate E2
 CALL addr (O)

In MiniTriangle, all constants and variables are *global*. Hence addressing relative to SB.

fetch [I] = LOAD d[SB] (where d is address bound to I relative to SB)
assign [I] = STORE d[SB] (where d is address bound to I relative to SB)

Elaboration must deposit value/reserve space for value on stack. Also, address of elaborated entity must be recorded (to be used by fetch and execute).

Code Optimization

Code Improvement

The code generated by a compiler *must* be correct and *should* also run fast, be small and use as little space as possible. Code improvement is the process of improving the time and space behaviour *without* changing the functional behaviour; i.e. correctness *must* be preserved.

Example: Replacing the code fragment $f(x) + f(x)$ by $2 * f(x)$ might improve the speed (saves a function call). However, *only* correct if f does *not* have any side effects:

Consider:

```
var x: Integer;
...
func f (y: integer): Integer ~
    begin
        x = x + 1;
        return x + y;
    end
...
x = 2;
putint(f(2) + f(2))
```

This code fragment would print 11, whereas the result of printing $2 * f(2)$ would be 10.

Code improvement usually referred to as *optimization*. However, it is hardly ever possible to *guarantee* optimality under any mathematical measure. It's not even always an improvement as it is not known what is going to happen at run-time, so we're optimizing for the *average expected* case.

Code improvement can be carried out at different levels: i) High level: source-to-source (AST) transformations, ii) Intermediate level: transformations on intermediate representation, e.g., "bare-bones" high-level language or control/data flow graph representation and iii) Low level: transformations on machine code.

Different levels make different kinds of information available such as High level for type information, Intermediate level for results of control/data flow analysis and explicit index calculations and bounds checks and Low level for result of code selection and register allocation.

Thus, each level is more or less suitable depending on the kind of optimization in question. A good optimizing compiler carries out optimizations at all levels.

Low-Level Optimizations

Typical examples of low-level optimizations include SPARC delay slots and static instruction scheduling; e.g. hiding read delays (requires data-flow analysis). 'Peephole optimizations' are *strictly local* improvement of short instruction sequences, e.g.

```
add %l1, 4, %l1  
sub %l1, 4, %l1
```

can be optimized to nothing!

Clearly, none of these low-level optimizations make any sense at a higher level!

Intermediate-Level Optimizations

Information that was implicit in the high-level representation might become explicit at the intermediate level, thus enabling/facilitating certain optimizations. Consider array indexing. High-level code fragments:

```
var x, y: array[1..100] Integer;  
...  
a = x[i] + y[i];
```

Intermediate (C-like) code with explicit pointer arithmetic:

```
if (i < 1 || i > 100) raise index_bounds;  
t1 = ^x + 4 * (i - 1);  
if (i < 1 || i > 100) raise index_bounds;  
t2 = ^y + 4 * (i - 1);  
a = t1 + t2
```

This could be optimized by reusing common subexpressions and eliminating redundant array bounds checks:

```
if (i < 1 || i > 100) raise index_bounds;  
t0 = 4 * (i - 1)  
t1 = ^x + t0;  
t2 = ^y + t0;  
a = t1 + t2;
```

Time vs. Space

Time and space optimizations are often in conflict. Consider representing an array of Booleans. If each Boolean represented by one machine word, we get fast access but it wastes space. If each Boolean represented by a single bit, this is more space efficient but access requires extra operations (shifting and masking); also takes space!

In other cases, small is fast as well:

Accessing memory is slow. The fewer instructions and the fewer pieces of data, the fewer memory accesses, and the faster the execution. It is highly desirable to keep inner loops small so that they fit in the first-level instruction cache. It is desirable to keep the set of 'currently accessed' memory locations small so that they fit in the first-level data cache.

But then again, since memory access is very slow, avoiding a memory access could sometimes sometimes be worth a few extra instructions as instruction fetching is typically much faster than data fetching because it is more predictable. The tradeoff between time and space is a highly complicated issue.

Compile-Time vs. Run-Time

Generating highly optimized code can take a *long* time! So the typical scenario is for a few optimizations to be turned on during development with more optimizations being turned on when compiling code to be distributed. Sometimes code *must* satisfy certain timing/space constraints. Such code might always have to be compiled with optimizations turned on.

Revision Notes

Many optimization techniques have quadratic or worse time complexity. Programs then usually optimized one part at a time, even though global optimization would yield better results. In some cases, spending days or weeks optimizing can make sense; e.g. compilation to hardware.

Common Optimization Techniques

Applicable at the source-code (AST) level and/or intermediate level:

1. Constant Folding:

Idea: evaluate (sub)expressions at compile-time where possible:

```
const pi ~ 3.1416;  
var volume, radius: Double;
```

...

```
volume = 4/3 * pi * radius^3;
```

4/3 * pi can be evaluated at compile-time:

```
const pi ~ 3.1415;  
var volume, radius: Double;
```

...

```
volume = 4.1888 * radius^3;
```

Not only applicable to *declared* constants:

```
x = 3;  
y = x + 1;  
x = x * 2;
```

can be optimized to

```
x = 3;  
y = 4;  
x = 6;
```

In general, *flow analysis* required:

```
x = 3;  
y = x + 1;  
while (x < z) begin  
x = x * 2;  
end
```

Unless z is known, we can only optimize to:

```
x = 3;  
y = 4;  
while (x < z) begin  
x = x * 2;  
end
```

One has to take care not to change the semantics. Compile-time floating point/integer precision must agree with run-time precision. Consider also cross compilation and rounding errors (e.g. dividing 1.0 by 3.0 might be more precise than loading a constant like 0.3333333333.), how to handle arithmetic exceptions, like division by 0.

2. Common Subexpression Elimination

Idea: avoid evaluating the "same expression" more than once.

```
x1 = y1 + 7 * z + 42;  
x2 = y2 + 7 * z + 42;
```

can be optimized to

```
t = 7 * z + 42;  
x1 = y1 + t;  
x2 = y2 + t;
```



Revision Notes

Common subexpressions often appear in *address computations* in intermediate code. The expressions must not only be *syntactically* the same; they must also *mean* the same thing: Scope rules must be taken into account:

```
let x = y * 17 in
let y = 13 in
let z = y * 17
```

The innermost $y * 17$ *cannot* be replaced by x .

Side effects must be taken into account (flow analysis):

```
x = y * 17 + 3;
y = y + 1;
z = y * 17 + 3;
```

Here, the two instances of $y * 17 + 3$ do *not* compute the same value. Indeed, the expressions themselves could have side effects:

```
x = y++ * 17 + 3;
z = y++ * 17 + 3;
```

3. Copy Propagation

Idea: After an assignment that *copies* a value, like $x = y$, use y in place of x wherever possible:

```
x = y;
v = x * 17;
w = x + 19;
```

can be transformed to

```
x = y;
v = y * 17;
w = y + 19;
```

It may then turn out that the assigned variable is *never used again*. In that case, the assignment is *dead code* and can be eliminated.

```
x = y;
v = y * 17;
w = y + 19;
```

can be optimized to

```
v = y * 17;
w = y + 19;
```

if x is never used again.

Copies often appear as a *result* of other optimizations, e.g. common subexpression elimination:

$t =$ *common expression*

```
x = t;
y = t + 10;
z = x + y;
```

4. Dead Code Elimination

Idea: It may be possible to *statically* determine that certain parts of the code will never be reached or will not have any effect. Such *dead* or *useless* code can be *removed* without changing the meaning of the program. Consider:

```
debug = false;
...
if (debug) System.out.println("Got here!");
```

After constant folding, we have

```
if (false) System.out.println("Got here!");
```

and the print statement is manifestly dead.

Assignment to variables that never get used is useless. Such assignments can be eliminated.



Recall the copy propagation example:

```
x = y;
v = y * 17;
w = y + 19;
```

can be optimized to

```
v = y * 17;
w = y + 19;
```

if x is never used again.

5. Strength Reduction

Idea: replace “expensive” operations by cheaper ones. Simple examples:

- Addition and shifting might be cheaper than multiplication:

$$5 * x \rightarrow x \ll 2 + x$$

- Multiplication might be cheaper than exponentiation:

$$x^2 \rightarrow x * x$$

$$z = x^5 \rightarrow x2 = x * x; z = x2 * x2 * x$$

Only applies when *known* integral power.

A loop may have a number of *induction variables* that remain in *lock step*:

```
i = 10;
while (i > 0) {
    i = i - 1;
    t = 4 * i;
    a[i] = b[t];
}
```

i and t are induction variables.

All that is going on is that t decreases by 4 each time round the loop. We can rephrase as follows:

```
i = 10;
t = 4 * i;
while (i > 0) {
    i = i - 1;
    t = t - 4;
    a[i] = b[t];
}
```

An potentially expensive multiplication has been replaced by a subtraction *inside* a loop.

6. Code Motion

Idea: code that is *loop invariant*, i.e. evaluate to the same value at each loop iteration, should be moved outside the loop.

```
for (i = 0; i <= m - 1; i++)
    for (j = 0; j <= n - 1; j++)
        x = x + a[i * 10 + j]
```

m-1 and n-1 invariant in the outer loop and i*10 invariant in the inner loop. Thus we can transform to:

```
t1 = m - 1;
t2 = n - 1;
for (i = 0; i <= t1; i++) {
    t3 = i * 10;
    for (j = 0; j <= t2; j++)
        x = x + a[t3 + j]
}
```

Array address computations and bounds checks often introduce loop invariant code fragments.



7. Algebraic Identities

Algebraic identities can be exploited to

- simplify expressions: $1 * x - 0 \rightarrow x$
- expose further opportunities for e.g. common subexpression evaluation:

$x = 2 + z;$

$y = z + 2;$

can be transformed to

$t = z + 2;$

$x = t;$

$y = t;$

However, must be careful! Computer arithmetic does not always obey the standard algebraic identities of mathematics! E.g. the order in which floating point numbers are added *does* matter:

if $x[i]$ is much less than y , then

$sum = 0;$

for ($i = 0; i < n; i++$) $sum = sum + x[i];$

$sum = sum + y;$

might not give the same result as

$sum = y;$

for ($i = 0; i < n; i++$) $sum = sum + x[i];$

8. Inlining

Idea: Avoid overhead of function/procedure call by instantiating the body with the actual parameters and copying the result to the call site. Also called *procedure integration*. Inlined procedures/functions should be *small* or size of code might blow up. Careful with *recursion*, otherwise the compiler might get stuck in a loop.

```
func f (x: Integer): Integer ~
begin
    return (x + 17) * 123;
end
...
x = f(a + 3);
y = f(x * 3);
```

Inlining would result in the last fragment becoming:

$x = ((a + 3) + 17) * 123;$

$y = ((x + 3) + 17) * 123;$

Consider:

```
func fib (x : Integer) ~ begin
    return (x < 2 ? x : fib(x-1) + fib(x-2));
end
```

If we blindly inline fib everywhere just because it initially looks small, we'll get stuck in a loop and the code size will blow up.



9. Interaction among Optimizations

One optimization might generate opportunities for other optimizations:

```
const level = 4;
const debugging = True;
func debug(severity: Integer) ~ begin
    return debugging && severity > level;
end
...
x = 10
if debug(3) then begin
    print "Oops! Well, got here.";
    x = x + 1;
end;
y = x + 10;
```

Inlining yields:

```
const level = 4;
const debugging = True;
...
x = 10;
if debugging && 3 > 4 then begin
    print "Oops! Well, got here.";
    x = x + 1;
end;
y = x + 10;
```

Constant folding yields:

```
x = 10;
if false then begin
    print "Oops! Well, got here."
x = x + 1;
end;
y = x + 10;
```

Dead code elimination yields:

```
x = 10;
y = x + 10;
```

And now we can do further constant folding!

```
x = 10;
y = 20;
```

And then, if x never used again, more dead code elimination: y = 20;

In general it is hard to pick a "best" order among the optimizations. Compilers often carry out optimizations iteratively until no further improvements can be made.