

# Compilation

---

Édition 16 January 2002

---



# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Compiler Structure	3
<b>2</b>	<b>Grammars</b>	<b>5</b>
2.1	Languages	5
2.2	Formal Grammars	5
2.3	Chomsky Hierarchy	5
2.4	Some Grammars	6
2.5	Regular Expressions	6
2.5.1	Finite Automata	6
2.6	Context-free Grammars	7
2.7	Ambiguous Grammars	8
<b>3</b>	<b>Lexical analysis</b>	<b>11</b>
3.1	Flex Expressions	11
3.2	Flex files	12
3.3	Rules	12
3.4	States	13
3.5	Semantic Values	14
3.6	Locations in the Scanner	14
3.7	Flex vs. Lex	14
<b>4</b>	<b>Syntactic Analysis</b>	<b>17</b>
4.1	Predictive Analysis	17
4.1.1	Intuition	17
4.1.2	Complications	18
4.1.3	Formalization	19
4.1.4	Grammar Massage	20
4.1.5	Application to Arithmetical Expressions	22
4.2	LR Analysis	25
4.2.1	LR(0)	25
4.2.2	SLR	26
4.2.3	LR(1)	26
4.2.4	LR(k)	26
4.3	Tunning a Yacc Grammar	27
4.3.1	Binary expressions	27
4.3.2	Handling the Dangling <code>else</code>	29
4.3.3	Unrolling rules	29
4.3.4	Error Recovery	31
4.4	Comparison between LR and LL analyses	31
4.4.1	Comparing LL and LR	31
4.4.2	Intellectual bigotry	32
<b>5</b>	<b>Abstract Syntax</b>	<b>35</b>
5.1	Symbols	35
5.2	Building the Parse Tree	35
5.3	Using Locations	36
5.4	Using Bison in C++	38

<b>6</b>	<b>Escapes</b> .....	<b>41</b>
	6.1 Escaping Variables .....	41
	6.2 Computing the Escapes .....	42
<b>7</b>	<b>Type Checking</b> .....	<b>43</b>
	7.1 Symbol Tables .....	43
	7.2 Types .....	43
<b>8</b>	<b>Frames</b> .....	<b>45</b>
	8.1 Stack Frames .....	45
	8.1.1 Frame Pointer .....	45
	8.1.2 Registers .....	46
	8.1.3 Call Conventions .....	46
	8.1.4 Variables Allocated on the Stack .....	46
	8.1.5 Static Link .....	46
	8.2 Examples .....	46
<b>9</b>	<b>Intermediate Representation</b> .....	<b>53</b>
	9.1 Tiger Intermediate Representation .....	53
	9.1.1 Expressions .....	53
	9.1.2 Instructions .....	54
	9.1.3 IR Samples .....	54
	9.1.3.1 R-value .....	54
	9.1.3.2 L-values .....	54
	9.1.3.3 Literals .....	55
	9.1.3.4 Control Structures .....	55
	9.1.4 Function Definition .....	56
	9.1.5 Translation to IR .....	56
	9.1.5.1 Translation Samples .....	56
	9.2 Normalization .....	57
	9.2.1 Normalization of ESEQ .....	58
	9.2.2 Normalization of CALL .....	58
	9.2.3 Normalization of CJUMP .....	58
	9.2.4 Finalization .....	59
<b>10</b>	<b>Instruction selection</b> .....	<b>61</b>
	10.1 Types des microprocesseurs .....	61
	10.2 Slection d'instructions .....	61
	10.2.1 Maximal Munch .....	61
	10.2.2 Dynamic Programming .....	62
	10.3 Implementation for Tiger .....	62
	<b>Index</b> .....	<b>63</b>

This document does not constitute by itself sufficient lecture notes for students to learn compilation. Nonetheless, there should be enough information for EPITA students to complete their project *provided* they assist to the lectures.

I'd like to thank Cyrille Belfort for his participation to this document, and for having translated some parts of the original French version.



# 1 Introduction

One of the first compiler, Fortran: 18 man-years. Tiger: 6 baby-months.

## 1.1 Compiler Structure

*Scanner* Group characters into *tokens*.

*Parser* Group tokens into sentences. Find the structure of the text.

*Semantic Actions*

Based on the structured reading, create an abstract syntax tree (AST).

*Semantic Analysis*

Link entity definitions to uses. Check the coherence (type checking, uses of `break`, escaping variables...). Heavy use of *symbol tables*.

*Frames, Activation Blocks*

Organisation des entités (variables etc.) sur la pile. Convention d'appel de fonctions.

*Translation*

Translating into an intermediate representation. End of the front end.

*Canonicalization*

Moving from a high level IR to a lower level IR.

*Instructions Selection*

First part of the back-end. Passage de la représentation intermédiaire vers un assembleur de haut niveau, mais spécifique à la machine cible (utilisation de (variables) *temporaires*).

*Analyse du flot de contrôle*

Quels sont les exécutions possibles ?

*Analyse de la vie des données*

Suivre la vie des temporaires de façon à ne les considérer que là où elles servent. Dissocier les noms inutilement communs.

*Register Allocation*

No temporaries.

*Code Production*

Finalisation, et production du code.



## 2 Grammars

### 2.1 Languages

A *language* is a set of *words*, or *sentences*.

Operations on languages: concatenation, alternation, power, closure.

Other operations: intersection, left and right divisions.

A parser, a scanner, more generally, an accepter are finite representation of a language.

### 2.2 Formal Grammars

Interesting languages are infinite. Problem: representing an infinite language as a finite entity.

An *alphabet*, a *string*, *concatenation* of strings, *power* of a string, *length* of a string, a *formal language*.

A *phrase structure grammar*, or *grammar*,  $G$  is a quadruple  $(T, N, S, R)$ ,  $T$  a set of terminal symbols,  $N$ ,  $N$  a set of non terminal symbols,  $S$ , the *axiom* is a non terminal, and  $R$  is a finite set of production rules. The vocabulary,  $V$  is the union of  $N$  and  $T$ .

A *rule* has the form ' $x \rightarrow y$ ' where  $x$  is a non empty string over  $V$  and  $y$  is a string over  $V$ .

A *proto-word* is a word made of terminals and non terminals.

Derivations.

Parse trees.

### 2.3 Chomsky Hierarchy

Type 0 Unrestricted phrase structure grammars, Turing Machine.

Type 1  
Context sensitive

Linear bounded Turing Machine. Lhs of the production rules must be shorter than lhs. Each production is of the form

$$\text{alpha1 A alpha2} \rightarrow \text{alpha1 beta alpha2}$$

where  $\text{alpha1}$  and  $\text{alpha2}$  are proto-words,  $A$  is a non terminal, and  $\text{beta}$  is a non empty proto-word, with one exception possible ' $S \rightarrow \text{epsilon}$ ' where  $S$  must not appear in a rhs.

Type 2  
Context free

Lhs must be a single nonterminal. Algebraic language. Any context free grammar can be rewritten without using the empty string in the rhs.

Type 3 Linear production rule: at most one non terminal in RHS.

Right linear production rules: ' $X \rightarrow aY$ ', ' $X \rightarrow a$ '. Right linear grammars: right linear rules plus ' $X \rightarrow \text{epsilon}$ '. Left linear rules, left linear grammars. Adding ' $X \rightarrow Y$ ' is OK.

What of grammars with right *and* left linear rules? What of ' $X \rightarrow aYb$ '?

Regular/rational languages, regular/rational expressions, finite state machines.

The inclusions are proper.

## 2.4 Some Grammars

Producing  $\{a^n b^n c^n \mid n \geq 1\}$  (From Salooma, ex. 2.1, p 11):

```

N = {S, A, B}
T = {a, b, c}
S = S
P = S   -> a b c
      S   -> a A b c
      A b -> b A
      A c -> B b c c
      b B -> B b
      a B -> a a A
      a B -> a a

```

Producing  $\{a^n b^n c^n \mid n \geq 1\}$  (From *Machines, Languages, and Computation*, ex. 3.1, p. 63):

```

S   -> A
A   -> a A B C
A   -> a b C
C B -> B C
b B -> b b
b C -> b c
c C -> c c

```

Producing  $\{a^n b^n \mid n \geq 1\}$  (From *Machines, Languages, and Computation*, ex. 3.2, p. 63):

```

S   -> A
A   -> a A B C
A   -> a b C
C B -> B C
b B -> b b
b C -> b

```

can you find a simple grammar defining the same language?

See Salooma, ex. 2.5, p 14 for a grammar of  $\{a^p \mid p \text{ prime}\}$ .

Declaration and uses in programming languages:

```
L = { w c w | w in {0, 1}+ }
```

## 2.5 Regular Expressions

Empty string

Symbol

Alternation

Concatenation

Kleene closure

Repetition

### 2.5.1 Finite Automata

DFA, NFA.

NFA: empty transition, multiple edges with the same label, multiple entry points.

Building a NFA corresponding to a regexp.

Determinization of NFAs.

## 2.6 Context-free Grammars

A grammar is *context free* if all the rules look like: ' $A \rightarrow \alpha$ ', with  $A$  as not terminal and  $\alpha$  is a proto-word.

Writing a C-like instruction sequence seems easy having '`ins`' and '`;`' as terminals..

```
S -> S ; S
S -> ins
S ->
```

Find a grammar producing the following words:

```
a := 1;
b := a + (c := a + 3, c)
```

First thing to do is obviously introducing terminals. In a computer science meaning terminals are not characters or data units, but of course lexical units. For the current grammar we choose '`id`', '`num`', '`+`', '`(`', '`)`', '`:=`' and '`;`':

```
id := num;
id := id + (id := id + num, id)
```

The only thing to do is to make a simple grammar.

```
S -> S ; S
S -> id := E
```

```
E -> id
E -> num
E -> E + E
E -> ( S , E )
```

Many different notations exist for grammars. First of them is the BNF, Backus-Naur Form. From '<http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html>'

BNF is an acronym for "Backus Naur Form". John Backus and Peter Naur introduced for the first time a formal notation to describe the syntax of a given language (This was for the description of the ALGOL 60 programming language, see [Naur 60]).

To be precise, most of BNF was introduced by Backus in a report presented at an earlier UNESCO conference on ALGOL 58. Few read the report, but when Peter Naur read it he was surprised at some of the differences he found between his and Backus's interpretation of ALGOL 58. He decided that for the successor to ALGOL, all participants of the first design had come to recognize some weaknesses, should be given in a similar form so that all participants should be aware of what they were agreeing to. He made a few modifications that are almost universally used and drew up on his own the BNF for ALGOL 60 at the meeting where it was designed. Depending on how you attribute presenting it to the world, it was either by Backus in 59 or Naur in 60. (For more details on this period of programming languages history, see the introduction to Backus's Turing award article in Communications of the ACM, Vol. 21, No. 8, august 1978. This note was suggested by William B. Clodius from Los Alamos Natl. Lab).

Since then, almost every author of books on new programming languages used it to specify the syntax rules of the language. See [Jensen 74] and [Wirth 82] for examples.

Uses of '`:=`' or '`::=`' instead of our '`->`' is the first difference. We uses the '`|`' of regular expresions, terminals are shown as '`this`', or bold characters..

```
S ::= S ';' S
    | 'id' ':=' E
```

```

E := 'id'
   | 'num'
   | E '+' E
   | '(' S ',' E ')'

```

Show out a BNF for floating point numbers

```

F := '-' FP
   | FP
FP := Cs
   | Cs '.' Cs
Cs := C
   | C Cs
C := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

```

- Parenthesis to group (compel priorities)
- '?' the character is optional(zero or one time).
- '\*' the character can be repeated as many time as you want(zero or more).
- '+' the character is not optional and can be repeated (1 or more)

Having such features help us to have more readable grammars, and eliminating recursion most of the time:

```

F := '-'? FP
FP := Cs ('.' Cs)?
Cs := C+
C := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

```

We can simplify even more:

```

F := '-'? C+ ('.' C+)?
C := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

```

Some people use '[A]' for '(A)?' and '{A}' for '(A)\*'.

## 2.7 Ambiguous Grammars

```

S := S ',' S | 'I'

```

Is this an ambiguous grammar? Yes it is: 'I; I; I' can be understood two different ways...It's an associativity problem; the same for instruction sequences. How can we deal with this problem?

```

S ::= S ; I | I

```

left recursive, or

```

S ::= I ; S | I

```

right recursive.

Ambiguity in context-free grammar can arise in two different ways:

- Some sentence has two different parse trees.

```

S -> S a S
S -> b

```

- Some sentence has two structurally equivalent parse trees, but the inner nodes are different.

```

A -> A a
A -> B a
A -> b
B -> B a
B -> b

```

For our little toy grammar :

$$E ::= E' + E \quad \text{or} \quad E ::= E + E'$$

$$E' ::= \text{id} \\ | \text{ num} \\ | (S, E)$$

The two are OK for addition but subtraction? '1 - 2 - 3' is '(1 - 2) - 3' or '1 - (2 - 3)'? The first one, of course, subtraction is left associative. Then use left recursion.

The simplest grammar for integer arithmetics:

$$E ::= E * E \\ | E + E \\ | E - E \\ | E / E \\ | ( E ) \\ | \text{ num}$$

it's ambiguous

Think about '1 + 2 \* 3' : sum of factors. To make it unambiguous we must give associativity and priority. First '()', then '\*', and '/', then '+' and '-'.

$$E ::= E + T \\ | E - T \\ | T$$

$$T ::= T * F \\ | T / F \\ | F$$

$$F ::= \text{ num} \\ | ( E )$$

This time it is not ambiguous, but it's the same language! Understanding that grammars are structures describing languages, but grammars are far more meaningful: by inference many grammars exist describing the same language (in fact an infinity, but it's not important because human can't imagine exactly this concept...unless someone can count till infinity i'll continue thinking that; it is not a shame, we are able to think about many concepts we can't catch up), good grammars, less good one (ambiguous is not good, I think you already understood that). The only thing to do is finding the good grammar matching what you need. That's why we will have much more interest in grammar properties than languages properties.



## 3 Lexical analysis

Regular languages allow to describe the tokens we will use. The “interesting” languages (such as arithmetics) are not regular, so we already know we will need more powerful techniques. Typically it means Lex will never be enough, we will need Yacc.

But then, since regular languages are context free languages (i.e., virtually, Yacc can do everything Flex does), why not using just Yacc. Simplify because of speed: FSM have more efficient implementations than FIXME: Finish this sentence!

### 3.1 Flex Expressions

There is one difference between the theory of finite state machine (recognizers), and those actually used in practice: the latter must recognize the *longest* match.

Here is the expressions supported by Flex. Note that this is more than plain regular expressions.

‘x’	match the character ‘x’
‘.’	any character (byte) except newline
‘[xyz]’	a "character class"; in this case, the pattern matches either an ‘x’, a ‘y’, or a ‘z’
‘[abj-oZ]’	a "character class" with a range in it; matches an ‘a’, a ‘b’, any letter from ‘j’ through ‘o’, or a ‘Z’
‘[^A-Z]’	a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.
‘[^A-Z\n]’	any character EXCEPT an uppercase letter or a newline
‘r*’	zero or more r’s, where r is any regular expression
‘r+’	one or more r’s
‘r?’	zero or one r’s (that is, "an optional r")
‘r{2,5}’	anywhere from two to five r’s
‘r{2,}’	two or more r’s
‘r{4}’	exactly 4 r’s
‘{name}’	the expansion of the "name" definition
“[xyz]\foo”	the literal string: ‘[xyz]foo’
‘\x’	if x is an ‘a’, ‘b’, ‘f’, ‘n’, ‘r’, ‘t’, or ‘v’, then the ANSI-C interpretation of \x. Otherwise, a literal ‘x’ (used to escape operators such as ‘*’)
‘\0’	a NUL character (ASCII code 0)
‘\123’	the character with octal value 123
‘\x2a’	the character with hexadecimal value 2a
‘(r)’	match an r; parentheses are used to override precedence
‘rs’	the regular expression r followed by the regular expression s; called "concatenation"
‘r s’	either an r or an s

' $\wedge r$ '	an $r$ , but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).
' $r\$\$$ '	an $r$ , but only at the end of a line (i.e., just before a newline).
' $\langle s \rangle r$ '	an $r$ , but only in start condition $s$
' $\langle\langle \text{EOF} \rangle\rangle$ '	an end-of-file

## 3.2 Flex files

Lex Directives

```
%{
```

Prologue

```
%}
```

Lex Directives

```
%%
```

Rules

```
%%
```

Epilogue

Le prologue et l'epilogue sont des sections de C.

La section des directives permet de

- rgler le comportement de lex (comme les options),
- dfinir les tats et les tats exclusifs,
- introduire des abrviations.

If the distinction between inclusive and exclusive start conditions is still a little vague, here's a simple example illustrating the connection between the two. The set of rules:

```
%s example
%%
```

```
<example>foo    do_something();
```

```
bar              something_else();
```

is equivalent to

```
%x example
%%
```

```
<example>foo    do_something();
```

```
<INITIAL,example>bar    something_else();
```

Without the ' $\langle \text{INITIAL}, \text{example} \rangle$ ' qualifier, the ' $\text{bar}$ ' pattern in the second example wouldn't be active (i.e., couldn't match) when in start condition ' $\text{example}$ '. If we just used ' $\langle \text{example} \rangle$ ' to qualify ' $\text{bar}$ ', though, then it would only be active in ' $\text{example}$ ' and not in ' $\text{INITIAL}$ ', while in the first example it's active in both, because in the first example the ' $\text{example}$ ' starting condition is an *\*inclusive\** (' $\%s$ ') start condition.

## 3.3 Rules

La forme gnrales des rgles est :

```
motif          action
```

L'exemple suivant implmente ' $\text{wc}$ ' :

```

%{
int num_lines = 0, num_chars = 0;
%}
%noyywrap
%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
int
main (void)
{
    yylex ();
    printf ("# of lines = %d, # of chars = %d\n",
           num_lines, num_chars );
}

```

Du point de vue de la priorité, lorsque plusieurs règles sont concurrentes, la liste est lue de haut en bas. Ceci est extrêmement utile lorsque l'on a des mots-clés et des identificateurs. L'action par défaut sur les caractères inconnus est de les propager vers 'stdout', donc toujours terminer par une règle attrape-miettes.

```

%{
enum token_e { VAR = 257, TYPE, INT, ID };
%}
%%
"var"      return VAR;
"type"     return TYPE;
[0-9]+     return INT;
[a-zA-Z][a-zA-Z_0-9]* return ID;
.          yyerror ("illegal token");

```

Les abréviations permettent d'alléger l'écriture des règles :

```

int      [0-9]+
id       [a-zA-Z][a-zA-Z_0-9]*
%%
"var"    return VAR;
"type"   return TYPE;
{int}    return INT;
{id}     return ID;
.        {
        yyerror ("illegal token");
        exit (1);
        }

```

### 3.4 States

On peut écrire des règles complexes pour les longs lexèmes tels que les chaînes de caractères et les commentaires. Ne pas oublier dans ces cas-ci que par défaut '.' ne reconnaît pas les retours à la ligne. Mais cette approche a plusieurs problèmes :

- Caractères d'échappement dans les chaînes de caractères.
- Déplacement de capacité du tampon de l'analyseur lexical (souvent 4K).
- Risque encore plus inutile dans le cas des commentaires, qui sont généralement purement et simplement jetés au niveau lexical.

La réponse à ces problèmes est l'utilisation d'états. Les états peuvent se comprendre comme des sous-analyseurs lexicaux. Quand on découvre l'ouverture d'une chaîne ou d'un commentaire, on bascule vers le sous-analyseur correspondant.

Les états se déclarent par la directive '`%x nom-de-l'état`'. L'état par défaut est 'INITIAL'. On utilise 'BEGIN' pour sauter d'un état à un autre. On utilise ensuite '`<nom-de-l'état>`' pour spécifier le bloc qui correspond au sous-analyseur.

```
%x STATE_COMMENT STATE_STRING
%%
"/*"      BEGIN STATE_COMMENT;

<STATE_COMMENT>{ /* Comments. */
  "/*"    BEGIN INITIAL;
  .|\n   ;

  <<EOF>> {
    yyerror ("Unexpected end of file in a comment");
    exit (1);
  }
}
```

### 3.5 Semantic Values

L'analyseur lexical doit reconnaître les catégories grammaticales des mots (ceci est le mot-clé 'begin', ceci est un espace, ceci est un identificateur). Dans certains cas, la catégorie n'est pas une information suffisante : dans le cas de chaînes de caractères, d'identificateurs, les littéraux etc. la *valeur* est également nécessaire.

Pour calculer la valeur d'un lexème sont disponibles 'yytext', un 'char \*' sur le début du lexème reconnu, et 'yyleng', sa longueur. La transmission de la valeur depuis l'analyseur lexical vers son client se passe par variables globales.

```
[0-9]+    the_value = atoi (yytext); return INT;
```

En règle générale le client de l'analyseur lexical est un analyseur syntaxique produit par 'yacc' ou équivalent. La convention est alors d'utiliser la variable globale 'yylval'. Comme on peut avoir plusieurs types de valeur, 'yylval' est une *union* :

```
%{
union
{
  int integer;
  char *string;
} yyval;
%}
{int}    yyval.integer = atoi (yytext); return INT;
{id}     yyval.string = strdup (yytext, yleng); return ID
```

### 3.6 Locations in the Scanner

### 3.7 Flex vs. Lex

```
<<EOF>>
Exclusive states
```

“Bugs” in the understanding of abbreviations.

Free.

Portable.

Stands as its own standard.

Generation of C++.



## 4 Syntactic Analysis

Most of the grammars we are interested in are not regular, but they are context free. This chapters presents the most classical techniques to parse (some) context free grammars.

### 4.1 Predictive Analysis

#### 4.1.1 Intuition

```
S ::= if E then S
    | begin S L
    | print string
```

```
L ::= end
    | ; S L
```

```
E ::= bool
```

Quelle est la propri t marquante de cette grammaire ? R flechir sur

```
if true then
begin
    print "Success";
    print "Suc s"
end
```

(notice that ‘;’ is a separator (like in Pascal), not end of instruction (C). Experience shows us that less programming errors are made when using c-like syntax, than separator. All those psychological aspects should be well considered before making the syntax of a language.

We always know in which case we are just looking the next word, because every rule begin with a non ambiguous key word.

```
enum token_e token;
```

```
void
S (void)
{
    switch (token)
    {
        case IF:
            accept (IF); E (); accept (THEN); S ();
            break;
        case BEGIN:
            accept (BEGIN); S (); L ();
            break;
        case PRINT:
            accept (PRINT); accept (STRING);
            break;
        default:
            error (token);
    }
}
```

using C++ can provide even more flexibility:

```

template <int k> class LL_k
{
private:
    vector<token>  tokens(k);
    exp            absyn;

    exp accept();
public:
    exp parse();
}

```

where `accept()` can accept the vector of token and give an `exp` emptying the vector, and `parse()` chooses the rule corresponding to the list of token, if no rule matches or if there is an ambiguity the next token is added to `tokens` and then retry other rules...

This method is called LL(1): Left to right, Leftmost Derivation, 1 look ahead token (we can use `k` tokens, in wich case it is called LL(`k`))

### 4.1.2 Complications

Does it always work when using

```

S ::= I
    | begin S L
    | print string

```

```

I ::= if E then S

```

```

L ::= end
    | ; S L

```

```

E ::= bool

```

and for

```

S ::= I
    | begin S L
    | print string

```

```

I ::= I1 I2

```

```

I1 ::=

```

```

I2 ::= if E then S

```

```

L ::= end
    | ; S L

```

```

E ::= bool

```

What happens if we try with our non-ambiguous arithmetics expression grammar

```

E ::= E + T      T ::= T * F      F ::= num
    | E - T      | T / F          | ( E )
    | T          | F

```

```

void
E (void)
{
  switch (token)
  {
    case ???:
      E () ; accept (PLUS) ; F () ;
      break;

    etc.
  }
}

```

### 4.1.3 Formalization

Calculating FIRST, we need to know the NULLABLE tokens and the FOLLOWING ones (we will use the word CANCELABLE and FOLLOW to simplify reading).

Algorithm upon production rules as  $A ::= A_1 A_2 \dots A_n$

```

All is initialized to empty
For every terminal A
  FIRST (A) := {A}
Repeat
  For every rule  $A ::= A_1 A_2 \dots A_n$ 
    If every  $A_i$  are NULLABLE or  $i = 0$ 
      Then A is NULLABLE
    For every i between 1 and n
      If  $A_1 \dots A_{i-1}$  are NULLABLE
        Then add  $A_i$  in FIRST(A)
      If  $A_{i+1} \dots A_n$  are NULLABLE
        Then add FOLLOW(A) to FOLLOW( $A_i$ )
      For every j between i+1 and n
        If  $A_{i+1} \dots A_{j-1}$  are NULLABLE
          Then add FOLLOW( $A_j$ ) to FOLLOW( $A_i$ )
      For end j;
    For end i;
  For end rule;
Until FIRST FOLLOW and NULLABLE don't change

```

Try with the following grammar

```

A ::= B
   | a

B ::= b
   |

C ::= A B C
   | c

```

we find

	Ann.	Pre.	Suiv.
A	oui	ab	abc

```

-----+-----+-----+-----
B      |oui      |b      |abc
-----+-----+-----+-----
C      |non      |abc    |(vide)

```

Writing functions (which are switch on current token), needs, for every function, knowing what to do for every token. More clearly (some people needs more explanation I feel it) we need an array (non terminal/terminal) wich contain rule to execute. ("If it look like an automaton, if it tastes like an automaton, if it sounds like an automaton, it surely can be used as an automaton" B. Franklin the mad with the thunder bolt, no it's not a citation but it is just to "open your eyes, open your mind[...] nescafee" sorry someone possessed my poor brain.)

For instance, it is clear that the rule 'A ::= a' has to be written in the cell (A, a). 'C ::= ABC' obviously has to be put in (C, a), but intuition also reveals that since 'A' is nullable, it has to be put in (C, b) too because b is member of FIRST(B). It is easy to define FIRST(ABC) and understand that 'C ::= ABC' is to be written in '{C} x FIRST(ABC)'.

What's to be done with 'B ::= '? Just think of parsing 'ac' with 'C' being the axiom: you want to use 'C -> ABC', and have 'B' disappear before the 'C' which will become 'c'. And in fact, since FIRST(C) is {a, b, c}, any 'B' before a 'C' should be ready to disappear: put it in (B, a), (B, b), (B, c).

What's to be done of 'A ::= B'? Imagine we analyse this text: 'c', 'c' as an axiom. According to (C, c), let's apply 'C ::= ABC'. We treat 'A' with 'c' as look-ahead. We feel that 'A' must become 'B', it's the only way for him to erase himself. We choose the rule 'B', with 'c' as look-ahead. As for 'A', 'B' must erase: 'B ::= ', i.e we now are at 'A.BC' with 'c' as look-ahead. 'B' erases: 'AB.C' with 'c' as look-ahead, we have exactly what we want!! We are the champions... Hence, 'A ::= B' dans FOLLOW(A), i.e., (A, a), (A, b) et (A, c).

The algorithm is:

A ::= A1 .. An

1. If a is member of FIRST(A1 .. An), put the rule in (A, a).
2. IF A1..An is nullable, put the rule in every (A, b) with b in FOLLOW(A)

Dans notre exemple, on obtient :

```

          a                b                c
-----+-----+-----+-----
A      A ::= a          A ::= B
      A ::= B          A ::= B          A ::= B
-----+-----+-----+-----
B
      B ::=
      B ::=          B ::=          B ::=
-----+-----+-----+-----
C      C ::= ABC      C ::= ABC      C ::= ABC & C ::= c

```

Where rules selected by rule (1) are on first line, and the other selected by (2) on the second line.

We notice that this grammar gives many choice for a rule: we can not analyse it with this method. Analysable grammar with this method are called LL(1) (like the parsing method of the same name, it means an LL(1) grammar can be parsed by an LL(1) parser). Generalysing for LL(k), only need to construct FIRST with k-uples of terminals...

Actually it's even worse than this: this grammar is ambiguous. Prove it, and exhibit a short word which can be parsed in an infinite number of ways.

#### 4.1.4 Grammar Massage

When trying to build a recursive descent parser for arithmetics, we find several problems.

```

E ::= E + T      T ::= T * F      F ::= num
   | E - T      | T / F          | ( E )
   | T          | F

```

First of all, this grammar is left-recursive:

```

E ::= E - T
   | T

```

which means that the routine which parses E's will look like:

```

void
E ()
{
  with (lookahead)
  {
    case ???:
      E (); accept (tok_minus); T (); break;
    case ???:
      T (); break;
  }
}

```

Obviously we enter an infinite loop (E calls itself) as soon as the first `case` is selected.

A grammar which includes a left recursive rule cannot be LL. So we have to make it right recursive :

```

E ::= T - E
   | T

```

Of course this implies that our “reading” is incorrect (here we read ‘-’ as if it were right associative), and we will have to adjust our semantic actions so that the proper order is respected.

We then face a second problem: how can we decide between ‘E ::= T - E’, and ‘E ::= T’? They all start the same way, i.e., we have:

```

void
E ()
{
  with (lookahead)
  {
    case FIRST (T):
      E (); accept (tok_minus); T (); break;
    case FIRST (T):
      T (); break;
  }
}

```

the two `case`'s are equal!

This issue is similar to the famous ‘if-then-else’:

```

I ::= if E then I
   | if E then I else I

```

We obviously can't make any switch that make us able to have this rule done correctly. Answer is simple: *left factoring*.

```

I ::= if E then I ELSE-OPT

ELSE-OPT ::= else I
           |

```

In the case of our arithmetical expressions, we'd use:

$$E ::= T E'$$

$$E' ::= - T E'$$

$$|$$

#### 4.1.5 Application to Arithmetical Expressions

First, the grammar must not be left recursive:

$$E ::= T - E$$

$$| T$$

$$T ::= F / T$$

$$| F$$

$$F ::= \text{num}$$

$$| ( E )$$

then we left factor it:

1.  $E ::= T E'$

2.  $E' ::= - E$
3.  $|$

4.  $T ::= F T'$

5.  $T' ::= / T$
6.  $|$

7.  $F ::= \text{num}$
8.  $| ( E )$

$E$  and  $E'$  are mutually recursive.  $E'$  can be rewritten to be independent from  $E$ , and similarly for  $T'$ :

0.  $S ::= E \$$

1.  $E ::= T E'$

2.  $E' ::= - T E'$
3.  $|$

4.  $T ::= F T'$

5.  $T' ::= / F T'$
6.  $|$

7.  $F ::= \text{num}$
8.  $| ( E )$

last, we need a means to know we reached the end of a file. We will use '\$' to denote the end of file, and add one rule:

0.  $S ::= E \$$

Then you get:

	Null.	First	Follow
S	n	( N	
E	n	( N	) \$
E'	o	-	) \$
T	n	( N	) \$ -
T'	o	/	) \$ -
F	n	( N	) \$ - /

hence

	(	)	-	/	N	\$
S	0				0	
E	1				1	
E'		3	2			3
T	4				4	
T'		6	6	5		6
F	8				7	

that can be make trivially by:

```

exp_t *
p_E (FILE *s)
{
    exp_t *t1, *t2;

    switch (token)
    {
        case tok_num:
            t1 = p_T (s);
            t2 = p_Ep (s);
            if (t2)
                return exp_bin_new (op_minus, t1, t2);
            else
                return t1;
            break;

        default:
            assert (0);
    }
}

```

```

exp_t *
p_Ep (FILE *s)
{
    exp_t *t1, *t2;

    switch (token)
    {
        case tok_minus:
            eat (s, tok_minus);
            t1 = p_T (s);
            t2 = p_Ep (s);
            if (t2)
                return exp_bin_new (op_minus, t1, t2);
            else
                return t1;
            break;

        case tok_eof:
            break;

        default:
            assert (0);
    }
}

```

Easy, isn't it? But JEDI says "easy, quick,...the best way to the dark side" And that's exactly it! The result is completely false: operators are considered right associative..

We must do left-recursivity but respecting operators associativity:

```

exp_t *
p_E (FILE *s)
{
    exp_t *t1;

    switch (token)
    {
        case tok_num:
            t1 = p_T (s);
            return p_Ep (s, t1);
            break;

        default:
            assert (0);
    }
}

```

```

exp_t *
p_Ep (FILE *s, exp_t *pred)
{
    exp_t *t1, *t2;

    switch (token)
    {
        case tok_minus:
            eat (s, tok_minus);
            t1 = p_T (s);
            t2 = exp_bin_new (op_minus, pred, t1);
            return p_Ep (s, t2);
            break;

        case tok_eof:
            return pred;
            break;

        default:
            assert (0);
    }
}

```

Unrecursiving that, allows us to obtain something simpler and more efficient:

```

exp_t *
p_exp (void)
{
    exp_t *res = p_term ();

    while (token == tok_plus || token == tok_minus)
    {
        if (token == tok_plus)
        {
            eat (tok_plus);
            res = exp_bin_new (op_plus, res, p_term ());
        }
        else /* token == tok_minus */
        {
            eat (tok_minus);
            res = exp_bin_new (op_minus, res, p_term ());
        }
    }

    return res;
}

```

## 4.2 LR Analysis

### 4.2.1 LR(0)

0.  $S ::= E \$$
1.  $E ::= '( L )'$
2.     |  $'x'$
3.  $L ::= L ', ' E$
4.     |  $E$

Try to parse  $'(x, (x, x, x))'$  by hand.  
Build the automaton.

### 4.2.2 SLR

Try to build the LR(0) automaton for

1.  $E ::= 'x' '+' E$
2.     |  $'x'$

	x	+	\$		E
1	s3				2
2			acc		
3	r2	s4/r2	r2		
4	s3				5
5	r1	r1	r1		

there is a conflict: this grammar is not LR(0).

What is the meaning of  $(x, 3) = r2$ ? It says "when I want to reduce to an E and the next token is 'x', reduce to E". But it is impossible to have an 'x' after an E. Why?

Simply because 'x' does not belong to FOLLOW(E).

To improve our table, we will put the reduction only when the token belongs to FOLLOW of the nonterminal being reduced. Since  $\text{FOLLOW}(E) = \{\$\}$  one gets:

	x	+	\$		E
1	s3				2
2			acc		
3		s4	r2		
4	s3				5
5			r1		

### 4.2.3 LR(1)

### 4.2.4 LR(k)

The book by Robin Hunter, "Compilers", 5.2, says that any LR(k) *language*,  $k > 1$ , is also a LR(1) language, and even a LR(0) language if each sentence is followed by an end marker.

However, there are for example LR(2) grammars which are not LR(1), say

```
%token a, t
%%
S:  F ', ' S
   | F

F:  a L

L:  t
```

```
| t ',' L
```

which in Bison produces the error ‘contains 1 shift/reduce conflict’.

The above grammar can be transformed into a LR(1):

```
S: a t F
```

```
F: /* empty */
   | ',' I F
```

```
I: a t
   | t
```

(Or so loc.cit. says.)

So Bison is just LR(1) or LALR(1) then, with no grammar transformation capabilities. – Loc.cit. suggests to first try say the simpler LALR(1) algorithm, and in the cases it fails, to use the LR(1) algorithm; but it does not cite an example of a LR(1) grammar which isn’t LALR(1), so I cannot check if Bison does that, or is simply LALR(1).

### 4.3 Tuning a Yacc Grammar

First thing first, always use Bison, drop Yacc dead. Secondly, when Bison complains because of conflicts, use ‘bison -v’ so that it dumps a file with a verbose description of the grammar and its possible problems.

If this is not enough, reduce the grammar to the core of the problem. Also, forget about all the problems of typing, i.e., keep only tokens. It is a pity that there are no option ‘--check’ that does this, hence you will probably need to strip the actions from the grammar.

#### 4.3.1 Binary expressions

Of course, you should use precedence to help Yacc. It is tempting to write

```
%token INT PLUS TIMES
%left PLUS
%left TIMES
%%
exp: INT
    | exp op exp;
op: PLUS | TIMES;
```

but you get 2 shift/reduce conflicts. In fact, you don’t even need two operators to get conflicts:

```
%token INT PLUS
%left PLUS
%%
exp: INT
    | exp op exp;
op: PLUS ;
```

gives 1 shift/reduce conflict:

```
exp -> exp . op exp   (rule 2)
exp -> exp op exp .   (rule 2)
```

```
PLUS      shift, and go to state 3
PLUS      [reduce using rule 2 (exp)]
$default  reduce using rule 2 (exp)
```

```
op        go to state 4
```

if you recall the way the associativity and the precedence are used to build the automaton, you should recall that we are actually comparing **a rule against a token**, not just two tokens. Here, the associativity directive helps to compare the rule ‘op: PLUS;’ against the token ‘PLUS’, which never happens! The problem is ‘exp: exp op exp’ against itself, you have to express the associativity of the rule itself:

```
%token INT PLUS
%left PLUS
%%
exp: INT
    | exp op exp %prec PLUS;
op: PLUS ;
```

here you transferred the associativity and precedence of ‘PLUS’ to the binary operator rule. Actually, the proper writing is:

```
%token INT PLUS
%left PLUS
%%
exp: INT
    | exp PLUS exp;
```

Now you should understand why, although it might look stupidly verbose, you must not factor the binary operators to a rule ‘exp: exp op exp’. You need:

```
%token INT PLUS TIMES
%left PLUS
%left TIMES
%%
exp: INT
    | exp PLUS exp;
    | exp TIMES exp;
```

the problem is that there is not enough context.

You might think the solution provided above to still use a nonterminal can be extended to bigger problems? Why not using several nonterminals to classify the operators per precedence, such as in

```
%token INT PLUS TIMES
%left ADDOP
%left MULOP
%%
exp: INT
    | exp mulop exp %prec MULOP;
    | exp addop exp %prec ADDOP;

mulop: TIMES;
addop: PLUS;
```

alas! It does not work. Can you explain why?

```
$ bison -dv arith.y
arith.y contains 4 shift/reduce conflicts.
$ cat arith.output
[...]
state 7
```

```
exp -> exp . mulop exp (rule 2)
exp -> exp mulop exp . (rule 2)
```

```

exp -> exp . addop exp (rule 3)

PLUS shift, and go to state 3
TIMES shift, and go to state 4

PLUS [reduce using rule 2 (exp)]
TIMES [reduce using rule 2 (exp)]
$default reduce using rule 2 (exp)

mulop go to state 5
addop go to state 6
[...]
```

If you can explain why, then you will easily find the right way to do it: it is possible.

### 4.3.2 Handling the Dangling else

One way is simply to tolerate the shift/reduce it produces. In this case, declare it to Bison:

```

%token INT IF THEN ELSE
/* Expected shift/reduce conflicts:
   - dangling else. */
%expect 1
%%
exp: IF exp THEN exp
    | IF exp THEN exp ELSE exp
    | INT;
```

Another way is to use precedence:

```

%token INT IF THEN ELSE
/* ELSE has higher precedence than THEN. */
%left THEN
%left ELSE
%%
exp: IF exp THEN exp
    | IF exp THEN exp ELSE exp
    | INT;
```

In case of doubt, always check the output file.

```

state 5
  exp -> IF exp THEN exp . (rule 1)
  exp -> IF exp THEN exp . ELSE exp (rule 2)
  ELSE      shift, and go to state 6
  $default  reduce using rule 1 (exp)
```

here, when it sees an 'ELSE', it shifts, which is exactly what we need.

### 4.3.3 Unrolling rules

There is a problem with the following portion of the Tiger grammar:

```

%token ID LBRACK RBRACK OF
%%
exp: typeid LBRACK exp RBRACK OF exp
    | lvalue;
lvalue: ID
      | lvalue LBRACK exp RBRACK;
typeid: ID;

```

The Tiger grammar is making the difference between the identifier of a type, 'ID', and that of a value (variable or function), 'typeid'. This creates an ambiguity reported by Bison:

```

% bison minitiger.y -v
minitiger.y contains 1 reduce/reduce conflict.

```

Then, read the content of 'minitiger.output':

```

State 1 contains 1 reduce/reduce conflict.
Grammar
rule 1    exp -> typeid LBRACK exp RBRACK OF exp
rule 2    exp -> lvalue
rule 3    lvalue -> ID
rule 4    lvalue -> lvalue LBRACK exp RBRACK
rule 5    typeid -> ID
state 1
  lvalue -> ID .    (rule 3)
  typeid -> ID .   (rule 5)

  LBRACK      reduce using rule 3 (lvalue)
  LBRACK      [reduce using rule 5 (typeid)]
  $default    reduce using rule 3 (lvalue)

```

Bison says in state 1 with 'LBRACK' as look ahead, it can either reduce rule 3 or 5. I.e., the situation it describes is:

```
my_id . [
```

At this point, it does not know where 'my\_id' is a type id, or a value id. And indeed, it is impossible to know! It can be 'my\_id [1] := 1', 'my\_id' is a variable holding an array or 'my\_id [1] of int', 'my\_id' is a type.

On the other hand, it is very easy to know: because of rule 1, we perfectly know that 'my\_id' is a type id if there is an 'of' after the bracket. But we can't see it, it is one level higher.

To help Bison, we will *unroll* the rule that bugs us:

```

%token ID LBRACK RBRACK OF
%%
exp: ID      LBRACK exp RBRACK OF exp
    | lvalue;
lvalue: ID
      | lvalue LBRACK exp RBRACK;

```

This time we have a shift/reduce conflict, let's check it:

```

exp      -> ID . LBRACK exp RBRACK OF exp    (rule 1)
lvalue   -> ID .                               (rule 3)

LBRACK   shift, and go to state 3
LBRACK   [reduce using rule 3 (lvalue)]
$default reduce using rule 3 (lvalue)

```

the parser will privilege shifting, i.e. it will prefer rule 1, which is not fine... It means it will misunderstand 'id [1] := 1'. We also need to unroll the other part!

```

%token ID LBRACK RBRACK OF
%%
exp: ID      LBRACK exp RBRACK OF exp
    | lvalue;
lvalue: ID
       | ID LBRACK exp RBRACK;

```

but of course this is dead wrong: since we killed the recursion, we no longer can handle ‘a[1][2]’. We must reintroduce the recursion, but only for long enough values:

```

%token ID LBRACK RBRACK OF
%%
exp: ID      LBRACK exp RBRACK OF exp
    | lvalue;
lvalue: ID
       | lvalue.big
lvalue.big: ID LBRACK exp RBRACK
           | lvalue.big LBRACK exp RBRACK;

```

This time, Bison accepts the grammar without any conflict.

### 4.3.4 Error Recovery

By default, if the grammar is not prepared to support syntax errors, the parser will stop on the first error.

There is a special token ‘error’ which can be used to specify how to “eat” syntax errors:

```

exp: LPAREN RPAREN
    | LPAREN exps RPAREN
    | LPAREN error RPAREN;

```

If there is an error “inside” ‘exps’, the parser will essentially pop everything which was on the state after the ‘LPAREN’, then discard all the input until it sees the ‘RPAREN’, then reduce this rule.

As you can see, the localization of the error handler is crucial. As a thumb rule, you should place your error token before a token which closes a construct. Another frequent use is in lists:

```

/* One or more ‘exp; ...’. */
exps.1:
    exp
  | exps.1 SEMI exp
  | error SEMI exp
;

```

## 4.4 Comparison between LR and LL analyses

### 4.4.1 Comparing LL and LR

<http://compilers.iecc.com/comparch/article/92-05-059>.

```

Newsgroups:  comp.compilers
From:        parrt@ecn.purdue.edu (Terence J Parr)
Keywords:    LL(1), LR(1), parse
Organization: Compilers Central
Date:        Mon, 11 May 1992 02:44:23 GMT

```

tiller@solace.me.uiuc.edu (Mike Tiller) writes:

*Is there anything to be cautious of when using LL? I looked in Principles of Compiler Design, but I couldn't decipher exactly what the pros and cons were.*

There is no strict ordering between LL(k) and LALR(1); that is, I know of one grammar that is LL(k), but not LALR(1). *However*, LALR(1) has stronger recognition strength in practice. yacc's advantage in this area has narrowed due to PCCTS's full LL(k>1) parsing (only k=1 was commonly available before). Strength is not the end of the story because one rarely wants to merely recognize → we *translate*. Here, LL parsers are the clear winner. A few highlights:

#### Mid-rule actions

Actions cannot introduce ambiguities into your LL grammar. All Yacc programmers say "arrgggghh!" here.

#### Inherited attributes

LL rules may inherit attributes; i.e. you can pass stuff to them just like in a programming language. For example, you can pass a scope to your rule that recognizes declarations. Future versions of PCCTS will even let you pass productions to rules → context-sensitive parsing.

#### Local variables

If recursive-descent compilers are generated, then local, stack-based variables are trivial to implement. Bottom-up folks have no *convenient* way. Local variables are useful because you get a new one at each rule invocation; e.g. a new 'scope' variable appears every time you enter rule **function**.

The disadvantage really only lies in the fact that you have more constraints when writing LL(k) grammars—no left-recursion and no common prefixes of tokens  $\geq k$  tokens in length.

*Basically, I just use Yacc (Bison whatever) to parse some input specification files.*

If you can do what you need to do with Yacc and/or already have grammars that do what you want, then there is no reason to change.

Moral of story: Most languages can be described easily with LL(k); the semantic flexibility of LL(k) makes any fancy footwork regarding the grammar worth the effort for me.

One other point: the tool's programming interface is also a consideration. Does the system allow EBNF? Can it build trees for you automatically? Can you debug the resulting parsers (tables vs recursive-descent) easily?

Terence Parr, parrt@ecn.purdue.edu Purdue University Electrical Engineering

## 4.4.2 Intellectual bigotry

<http://www.google.com/url?sa=U&start=4&q=http://compilers.iecc.com/comparch/article/95-11-138&e=42>

```
Re: LL(1) vs LALR(1) parsers
Newsgroups:    comp.compilers
From:          "steve (s.s.) simmons" <simmons@bnr.ca>
Keywords:      parse, LALR, LL(1)
Organization:  Bell-Northern Research Ltd.
References:    95-11-051
Date:          Wed, 15 Nov 1995 13:39:07 GMT
```

*Why do compiler design text book authors only describe recursive parsers as a step stone on their way to bottom up parsers? I know that LALR allows for*

*more complex grammars, but many languages can be described by  $LL(1)$  grammars. If  $LL(1)$  can yield faster, smaller and more understandable parsers with better error handling for many languages, why isn't this method more elaborated? (Holub, as an example, presents full grammar and compiler listings for a bottom-up C-compiler, not for top-down).*

Intellectual bigotry!!!!

That is, the automated parsers use a great deal of automata theory which helps build on the computer science foundation. Recursive descent parsers are a small matter of programming (SMOP). I do notice among peers in industry that people are no longer snubbing the idea of writing a recursive parser when it is appropriate.

Remember you don't take a compiler course to learn how to build a compiler, surprise... Most people never write a compiler. You take it for the following reasons:

- Improving your comp. sci. background to understand automata theory with a very good application.
- Understanding what a compiler may do (or not do) for your code.
- Learning about big system software, data structures, etc.

Thank you.

Steve Simmons

[I agree that few CS students will write a conventional compiler, but we all end up having to decode some sort of input language in an application. -John]



## 5 Abstract Syntax

The abstract syntax is one of the several intermediate representations which are used in compilers. It represents the input text without all the punctuation such as ‘{’, ‘}’, ‘,’, ‘;’ etc. which have no computation meaning, but which are required by the *concrete* syntax to express the structure of the text.

Now that the text is parsed, the structure is simply given by the tree itself.

This section considers only the case of Yacc parsers.

### 5.1 Symbols

Obviously, when the scanner meets a string or an integer, it “returns” a string or an integer. What should it do about identifiers.

There are several solutions, the most obvious one being just returning the string which composes the identifier. But this is very inefficient: you will allocate a new string each time you meet an identifier, even if you see the same one a million times, you need to ‘strcmp’ to check whether two identifiers are equal etc.

An excellent alternative, which is widely used in computer science, consists in mapping all the strings which are equal to a unique one. In Tiger, this unique string is named a *symbol*.

### 5.2 Building the Parse Tree

We will use the action and the semantic values of the Yacc parsers in order to generate the parse tree. To this end we first have to augment the definition of the ‘%union’ with all the types of information we will need while building the tree. Something like:

```
%union yystype
{
  /* Tokens. */
  int ival;
  char *str;
  symbol_t *symbol;

  /* Non terminal. */
  var_t *var;
  ty_t *ty;
  field_t *field;
  exp_t *exp;
}
```

then you precise what are the type of the semantic values of the non terminals:

```
%type <var> lvalue biglvalue
%type <exp> exp
%type <ty> ty
%type <field> tyfield
```

and add the right actions to each rule:

```
exp:
  INT      { $$ = exp_int_new ($1) }
| STRING  { $$ = exp_string_new ($1) };
```

## 5.3 Using Locations

*This section was co-written by Cyrille Belfort.*

When the abstract syntax tree is built, the validity of the program being compiled is not yet guaranteed, in particular type checking is yet to be performed. Therefore, since we have to report accurate error messages to the user, we must find a means to keep track of the locations of the tokens, and of the extend of larger constructs.

Bison provides such a service, via so called *locations*. It keeps track for you of the extend of constructs. Unless specified by the user, it uses an aggregate which contains:

```
int first_line;
int first_column;
int last_line;
int last_column;
```

Of course you need to tell your parser the location of the tokens, which you do via ‘`yylloc`’, as you do for semantic values with ‘`yylval`’. This phase is delicate and it is a pity that Flex doesn’t provide a direct support for this.

We can use C++ class to implement locations.

With much intellectual work and some intuition, using a class which provides tools helping manipulate one line and one column seems to be interesting: calling this class `Position` shows that we are next to sanity. Some hours later our guardian angel appears to us ostentatiously and tells us: “`Position` is good but something is missing, your tokens should have a beginning position but none without an end indicator”. That is the reason why we did a location. Let us show it to you:

```
class Position
{
public:
    //some usefull/useless constante to remmember
    //where does lines and columns begin

    static const int initial_line = 1;
    static const int initial_column = 1;

    //we should be able to construct the class
    //i think it wouldn't compile without a constructor

    Position() : filename ("none"),
                line (initial_line),
                column (initial_column)
    {
    }

    //an assignment operator

    const Position&
    operator =(const Position& pos)
    {
    //no you can do it by yourself
    {

        //column modification
```

```

inline
void
column_add(/*what could be here?*/)
{
//nothing could be easier
}

//line modification it's a litle bit harder: one more code line
inline
void
line_add(/*something's missing, isn't it?*/)
{
//a brain could help filling this
}

//did you notice we worked without any column or line

public:
std::string filename;
int line;
int column;
}

```

You can add operators like '+', '+=', '<<' but be carefull you shouldn't forget there are columns and lines.

So Position is good for healthy people. Now here is a Location example

```

class Location
{
public:

//this is a construtor we did it for you because
//it's special: it calls on other construtors

Location() : begin(),
            end()
{
}

Position begin;
Position end;

//we will only tell you what should be used to help:
//a step : begin = end
//column adds to end
//line adds to end
//word begin = end and end+ = 1
}

```

As usually you can (and should) add some operators : '+=', '+', '<<'.

With those classes you are ready to set good locations in your scanner. But remember: nothing is error safe so do it **carefully**. Here is a (simple) use of location. YYLTYPE is Location, so yylloc is a Location.

```
extern YYLTYPE yylloc;
```

```

//some/many lines
//we only show it on some key words:

    "if" { yylloc.word(yyleng); return IF; }
"then" { yylloc.word(yyleng); return THEN; }

//for those who don't understand the word function : begin goes to
//last end, end goes to the end of the token

```

Now, in your Bison grammar file, you may use ‘@\$', ‘@1’, ‘@2’ etc. just as you use ‘\$\$’ and the like. In particular, each node of the abstract syntax tree should be decorated with its location, i.e.:

```

exp:
  NIL { $$ = new absyn::NilExp (@$); }
  | lvalue ASSIGN exp { $$ = new absyn::AssignExp (@$, $1, $3); }
  | IF exp THEN exp { $$ = new absyn::IfExp (@$, *$2, *$4); }

```

It is quite easy to check that you computed properly the locations: just verify that the location of the result of your parsing corresponds to the full program.

You can also specify your own structure (for C-like parsers), and even a (better?) C++ class actually. You must not forget the mandatory members which are expected by Bison:

To use this feature, you need to define the structure you want to use for locations, for instance:

```

struct position_s
{
  string filename;
  int first_line;
  int first_column;
  int last_line;
  int last_column;
};

```

then you teach Bison which structure it must use for locations: just ‘#define YYLTYPE struct position\_s’ in your parser. And of course, be sure that your scanner can “see” this definition too.

## 5.4 Using Bison in C++

Bison does not support C++ in the sense that it does not provide a reified interface. Nonetheless, since it produces C code, it is possible to use C++ code in your parser. Actually, you can just program as if C++ was supported.

Nonetheless, there is an important shortcoming due to the use of a C union to implement ‘%union’. You cannot write:

```
%union yystype
{
  /* Tokens. */
  int ival;
  string str;
  Symbol symbol;

  /* Non terminal. */
  Var var;
  Ty ty;
  Field field;
  Exp exp;
}
```

because it is illegal in C++ to put an object with a constructor into a union:

```
parsetiger.y:32: member 'class string yystype::str' with constructor \
      not allowed in union
```

nevertheless, it is allowed to use pointers to objects:

```
%union yystype
{
  /* Tokens. */
  int ival;
  string *str;
  Symbol *symbol;

  /* Non terminal. */
  Var *var;
  Ty *ty;
  Field *field;
  Exp *exp;
}
```

unfortunately this means that if you wanted to work this references, you will have to cripple your semantic actions with dereferences, and `new`:

```
exp:
  NIL
  { $$ = new NilExp (@$) }
| lvalue ASSIGN exp
  { $$ = new AssignExp (@$, *$1, *$3) }
| IF exp THEN exp
  { $$ = new IfExp (@$, *$2, *$4) }
```

You will notice that anyway in the case of 'Exp' and the like, since it is an abstract class, the C++ requires that you use a pointer.



## 6 Escapes

### 6.1 Escaping Variables

A language is said to support the *block structure* when all the nesting levels have the same rights. For instance C does not have this property since you are not allowed to introduce functions in functions, but only at the top level.

One of the most typical features of languages with block structure is the ability for nested functions to access the variables of the outer functions. Tiger being one such language, here is a dummy example of this sharing:

```
let function outer () =
  let var escaping := "Bye Bye!"
      var prisoner := "Boowoo..."
      function inner (msg:string) =
        (print (msg); print (escaping))
      in
        inner (prisoner)
      end
  in
    outer ()
  end
```

The variable `escaping` is used by a nested function, and this nested function will need an access to it. This variable is said to *escape*. The variable `prisoner` does not escape.

As we will see with more details later, escaping variables require a special treatment. For a start, you can easily imagine that we cannot put an escaping variable in a register<sup>1</sup>. As you can see, we need to know whether a variable escapes where it is defined: the IR translation will need to know where to allocate the newly declared variable (register or stack). Unfortunately the escaping status of a variable is not known until all its uses have been checked.

Therefore, we have to compute the escapes before performing the translation: these two phases cannot be done concurrently (well, sure you can, but that makes it hacky, hairy, fragile, and complex). You are free to compute them before or after type checking, they are commutative when there are no error in the source.

What if there are errors?

Obviously not only variables are concerned by this: formals too:

```
let function print_escaping (escaping: int, not_escaping: int) : int =
  let function escaping_value () : int = escaping
      in
        escaping_value () + not_escaping
      end
  in
    print_escaping (1, 2)
  end
```

Finally, note that the outer level, the top level, with this regard, should be considered as a function:

```
let var escaping      := "I rule the world!"
    var not_escaping := "Peace on Earth for humans of good will."
```

<sup>1</sup> Actually it is possible, but it would require inter function register allocation. Since we compile functions, nested or not, independently, we cannot.

```
    function print_escaping () = print (escaping)
in
  not_escaping;
  print_escaping ()
end
```

## 6.2 Computing the Escapes

The algorithm is dead simple: walk the abstract syntax tree, tag each variable declaration tag as “not escaping”, each time you see a use of a variable that was not declared with the current scope, tag its definition site as “escaping”.

This means we need a means to recover the definition site of a variable each time we find a use: we need a table of symbols. Obviously the boundaries we are interested in are related to the definition of functions, therefore, `scope_begin` and `scope_end` are invoked when visiting a function declaration.

You are encouraged to study the following example, which might exhibit something you did not consider...

```
let var public := 0
in
  /* Redefine PUBLIC to stress the escape checker. */
  let var public := 1 in public end;
  /* Now actually use PUBLIC as an escaping variable. */
  let function inner () : int = public
  in
    inner ()
  end
end
```

## 7 Type Checking

The main task of the semantic analysis, is to relate the occurrences of an entity, and to check they are used consistently. The most important aspect is *type checking*.

### 7.1 Symbol Tables

It is extremely easy to type check something like '1 + 2' or even '1 + "two"', since it is easy to know the type of a literal. But what can you do with 'a + b', where 'a' and 'b' are variables? You need to remember what was the type of 'a' and 'b' the last time you met them, i.e., you have to keep a database on the entities the user can define (types, variables, functions and so on).

Such databases are called *symbol tables*.

These databases are constantly changing. For instance in the following code:

```
let var i := 0
    function id (i : string) : string = i
in
    i := i + 1;
    id ("i")
end
```

there are three symbol table, or, if you prefer, the symbol table goes through three different states.

### 7.2 Types

There are basically two equalities for types:

#### *intensional equality*

Two types are considered equal if they are the same individual, i.e., rigorously the same type instance. This is the case in Ada, Pascal, Tiger etc.

#### *extensional equality*

Two types are considered equal iff their structures match. This is the case in C, C++ etc.

The following types

```
type int1_list = { hd: int, tl : int1_list }
type int2_list = { hd: int, tl : int2_list }
```

are intensionally different, but extensionally equal. While you may assign an object of type `int1_list` to a variable of type `int2_list` in C, it would raise a type violation in Tiger, Pascal etc.

Note that the *creation* of a type is not to be confused with the aliasing:

```
type int1_list = { hd: int, tl : int1_list }
type int2_list = int1_list
var list1 = int1_list { hd = 1, tl = nil }
var list2 = list1
```

is valid in Tiger.



## 8 Frames

Recursion requires dynamically allocated memory for local variables and parameters. The natural and efficient implementation of this memory is stack like.

### 8.1 Stack Frames

*Stack Frame*, or *Frame* for short, and *Activation Record* denote the same thing: the room reserved in the stack for the current function.

There are several things that needs to be stored in the stack:

arguments

local variables

aka automatic variables (automatically allocated when entering in the function).

return address

where to go back on 'return'.

saved registers

i.e., save the environment of the caller to be able to restore it.

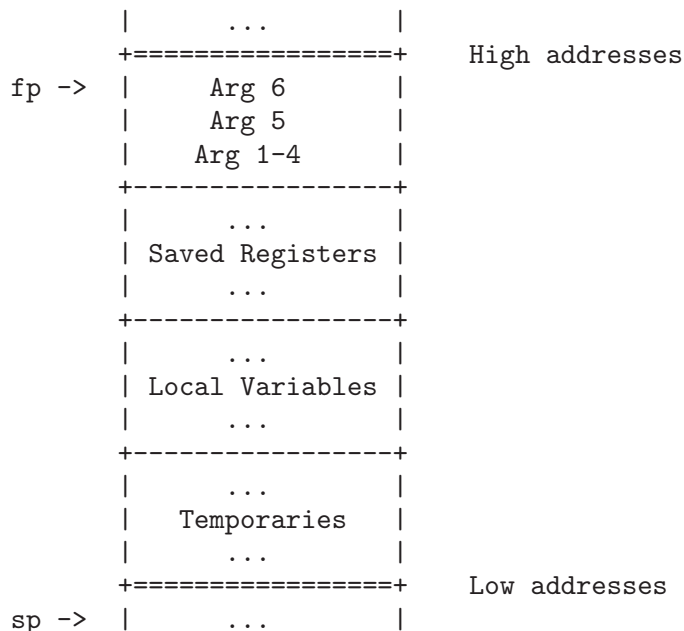
temp

i.e., automatic variables that the compiler needs when there are not enough registers.

static link when needed

Since most `call` and `return` instructions do not influence the stack, you are quite free with the order in which you store the various items. Nevertheless, it is desirable to follow the conventions given by the manufacturer, so that objects from different compilers are inter-operable.

On MIPS:



#### 8.1.1 Frame Pointer

As one can see, different functions have different storage needs, therefore we will need two informations: where the stack frame starts (the *frame pointer*, `fp`), and where it ends (the *stack pointer*, `sp`).

### 8.1.2 Registers

Modern architecture have a set of caller-save registers and callee-save registers. Of course this is just a convention: the microprocessor itself has no preference.

Suppose `f` uses `x` before calling `g`, and no longer needs it afterwards, where is it going to be stored?

Suppose `x` is used before calling several functions, and reused afterwards, what's the best thing to do?

### 8.1.3 Call Conventions

Although you may have to save arguments on the stack, using registers saves time:

- Dynamically, more leaf functions than others. They can even avoid allocating a frame.
- Not all the registers need to be saved.
- Inter-procedural register allocation.
- SPARC's register windows.

### 8.1.4 Variables Allocated on the Stack

In particular those which escape.

### 8.1.5 Static Link

When `f` is called, it is given a pointer to the activation record of the current frame of the function that immediately encloses `f`.

## 8.2 Examples

```
static int
id (int x)
{
    int res = x;
    return x;
}

int
sum (int x, int y)
{
    int res = id (x + y);
    return res;
}

bash$ cc -O0 -S foo.c -O
bash$ cat foo.s
        .verstamp      2 11
        .text
        .align 2
        .file 2 "foo.c"
        .loc 2 3
```

```
# 1 static int
# 2 id (int x)
# 3 {
    .ent    $$3 2
$$3:
    .option 00
    subu    $sp, 8
    .frame  $sp, 8, $31
    .loc    2 4
# 4     int res = x;
    sw     $4, 4($sp)
    .loc    2 5
# 5     return x;
    move   $2, $4
    b     $32
    .loc    2 6
# 6 }
    b     $32
$32:
    addu   $sp, 8
    j     $31
    .end   id
    .text
    .align 2
    .file  2 "foo.c"
    .globl sum
    .loc   2 10
```

```

# 7
# 8 int
# 9 sum (int x, int y)
# 10 {
    .ent    sum 2
sum:
    .option 00
    subu   $sp, 32
    sw     $31, 20($sp)
    sd     $4, 32($sp)
    .mask  0x80000000, -12
    .frame $sp, 32, $31
    .loc   2 11
# 11    int res = id (x + y);
        lw     $14, 32($sp)
        lw     $15, 36($sp)
        addu   $4, $14, $15
        .livereg 0x800ff0e,0xffff
        jal   $$3
        sw     $2, 28($sp)
        .loc   2 12
# 12    return res;
        lw     $2, 28($sp)
        b     $33
        .loc   2 13
# 13    }
        b     $33
$33:
        lw     $31, 20($sp)
        addu   $sp, 32
        j     $31
        .end   sum

```

```

bash$ cc -S -O1 foo.c
bash$ cat foo.s
        .verstamp      2 11
        .text
        .align 2
        .file 2 "foo.c"
        .loc 2 3

```

```

# 1 static int
# 2 id (int x)
# 3 {
    .ent    $$3 2
$$3:
    .option 01
    subu   $sp, 8
    .frame $sp, 8, $31
    .loc   2 4
# 4     int res = x;
    sw     $4, 4($sp)
    .loc   2 5
# 5     return x;
    move   $2, $4
    addu   $sp, 8
    j      $31
    .end   id
    .text
    .align 2
    .file  2 "foo.c"
    .globl sum
    .loc   2 10
# 6 }
# 7
# 8 int
# 9 sum (int x, int y)
# 10 {
    .ent    sum 2
sum:
    .option 01
    subu   $sp, 32
    sw     $31, 20($sp)
    sd     $4, 32($sp)
    .mask  0x80000000, -12
    .frame $sp, 32, $31
    .loc   2 11
# 11     int res = id (x + y);
    lw     $14, 32($sp)
    lw     $15, 36($sp)
    addu   $4, $14, $15
    .livereg      0x800ff0e,0xffff
    jal   $$3
    sw     $2, 28($sp)
    .loc   2 12
# 12     return res;
    lw     $2, 28($sp)
    lw     $31, 20($sp)
    addu   $sp, 32
    j      $31
    .end   sum

```

```
bash$ cc -S -O2 foo.c
```

```

bash$ cat foo.s
        .verstamp      2 11
        .text
        .align 2
        .file 2 "foo.c"
        .loc 2 3
# 1 static int
# 2 id (int x)
# 3 {
        .ent    $$3 2
$$3:
        .option 02
        .frame $sp, 0, $31
        .loc 2 4
# 4 int res = x;
        .loc 2 5
# 5 return x;
        move   $2, $4
        j      $31
        .end   id
        .text
        .align 2
        .file 2 "foo.c"
        .globl sum
        .loc 2 10
# 6 }
# 7
# 8 int
# 9 sum (int x, int y)
# 10 {
        .ent    sum 2
sum:
        .option 02
        subu   $sp, 24
        sw     $31, 20($sp)
        .mask  0x80000000, -4
        .frame $sp, 24, $31
        move   $3, $4
        .loc 2 11
# 11 int res = id (x + y);
        addu   $4, $3, $5
        .livereg 0x800ff0e,0xffff
        jal   $$3
        .loc 2 12
# 12 return res;
        lw     $31, 20($sp)
        addu   $sp, 24
        j      $31
        .end   sum

```

In the following example, there are too many incoming arguments.

```

int
sum8 (int x1, int x2, int x3, int x4,
      int x5, int x6, int x7, int x8)
{
    return x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8;
}
bash$ cc -S foo.c -O2
      .verstamp      2 11
      .text
      .align 2
      .file 2 "foo.c"
      .globl sum8
      .loc 2 4
# 1 int
# 2 sum8 (int x1, int x2, int x3, int x4,
# 3      int x5, int x6, int x7, int x8)
# 4 {
      .ent sum8 2
sum8:
      .option O2
      .frame $sp, 0, $31
      .loc 2 5
# 5 return x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8;
      addu $14, $4, $5
      addu $15, $14, $6
      addu $24, $15, $7
      lw $25, 16($sp)
      addu $8, $24, $25
      lw $9, 20($sp)
      addu $10, $8, $9
      lw $11, 24($sp)
      addu $12, $10, $11
      lw $13, 28($sp)
      addu $2, $12, $13
      j $31
      .end sum8

```



## 9 Intermediate Representation

A compiler includes a lot of code which is independent from the source language, and the target language. Par conséquent, la factorisation est recherchée. Mais si on supporte  $s$  langages sources et  $c$  langages cibles, on doit implémenter  $s \times c$  traductions ???

Bien entendu on va chercher produire un langage intermédiaire, une représentation intermédiaire, ce qui permet de se limiter  $s + c$ . Remarque que la syntaxe abstraite est déjà une façon d'abstraire du langage de départ. Certains langages peuvent probablement être exprimés dans la même syntaxe abstraite, mais s'il existe autant de langages aujourd'hui c'est aussi et surtout parce qu'ils sont profondément différents, et par conséquent leur syntaxe abstraite sont aussi très différentes.

Pourquoi ne pas se définir une syntaxe abstraite universelle vers laquelle traduire tous les langages ?

D'abord parce que cette syntaxe abstraite n'a pas de raison d'exister, et si elle existait, on aurait les défauts suivants :

*syntaxe abstraite riche*

donc en aval, on aurait beaucoup de cas à traiter ;

*traduction difficile*

la traduction, au sortir du parseur, vers la syntaxe abstraite sera certainement plus délicate que pour une syntaxe abstraite spécialisée.

Par conséquent, en général, c'est un peu plus loin qu'on canonisera la représentation de langages différents : juste avant l'assembleur.

La représentation intermédiaire, sous une forme linéaire, est un bon candidat de byte-code.

### 9.1 Tiger Intermediate Representation

La représentation intermédiaire sera la dernière représentation entre la partie haute, proche du langage source, et la partie basse, proche du langage cible. On a donc tout intérêt à se rapprocher de notre langage cible. Comme ici on parle de compilation vers l'assembleur, on va prendre des instructions relativement élémentaires. Ceci dit, on va chercher rester assez générale : il s'agit de n'avoir qu'une représentation intermédiaire quelque soit les assembleurs qui suivent.

De toutes façons, on veut aussi qu'il ne soit pas exagérément difficile de passer de l'arbre abstrait vers la représentation intermédiaire.

Voici une proposition de RI.

#### 9.1.1 Expressions

'const( $i$ )'

'name( $n$ )' pour les noms comme les labels

'temp( $t$ )'

'binop(op,  $e_1$ ,  $e_2$ )'

pour les expressions arithmétiques, mais pas pour les booléennes qui seront traitées spécialement.

'mem( $e$ )'

'call(fun, args)'

'eseq( $s$ ,  $e$ )'

pour l'implémentation de la virgule comme en C.

### 9.1.2 Instructions

‘move(temp(*t*), *e*)’

‘move(mem(*e*<sub>1</sub>), *e*<sub>2</sub>)’

‘exp(*e*)’ valuer, jete

‘jump(*e*)’ Le paramtre *e* est valu *e* donne la destination (adresse, ou label).

‘cjump(*op*, *e*<sub>1</sub>, *e*<sub>2</sub>, *if\_true*, *if\_false*)’

‘*e*<sub>1</sub> *op* *e*<sub>2</sub>’ s’value en boolean. La raison c’est qu’on veut pouvoir avoir les oprations boolennes paresseuses.

‘seq(*s*<sub>1</sub>, *s*<sub>2</sub>)’

pour l’implmmentation de la virgule comme en C.

‘label(*n*)’

Note that this IR is weak: there is nothing to translate `switch` statements and many others. To represent `switch`, just extend `jump(e)` into `jump(e, labs)` where *labs* holds the list of the possible destination of the `jump`. This is required for liveness analysis.

Au total on a trois types : les expressions, les instructions, et les conditionnelles.

### 9.1.3 IR Samples

#### 9.1.3.1 R-value

Lecture d’une variable dans le frame d’une routine :

mem(binop(plus, const(*k*), temp(fp)))

qu’on peut se permettre d’abrger par

mem(+ (const(*k*), temp(fp)))

Ca peut devenir plus pnible pour accder une variable en descendant les static links.

mem(const(*k*) + const(*L*<sub>(*n*-1)</sub>) + ... + const(*L*<sub>1</sub>) + temp(fp))

where *L*<sub>*i*</sub> are the static links.

Pour un tableau, la descente de l’indice ncessite du code dynamique d’accs la variable de l’indice, puis conversion en adresse de la cellule du tableau. Bien entendu un bon compilateur est suppos prendre garde que l’on accde un indice valide (Bound Checking). Il est certain que tous les langages ne le permettent pas, en particulier le C ne rend pas la tche aise. Par contre en Ada, pas de problme.

Pour une structure, il faut ajouter l’offset.

#### 9.1.3.2 L-values

L’affectation d’un tableau un autre peut avoir plusieurs sens selon la smantique des langages. Qu’en est-il du Pascal et du C ?

C’est la mme chose pour les structures. Qu’en est-il du Pascal et du C ?

L’implmmentation aura peut-tre besoin de faire appel une recopie de blocs de mmoire : mem devrait avoir un autre paramtre pour spcifier la taille.

### 9.1.3.3 Literals

Pour les chaînes de caractères (et autres bitmaps), on se donnera une fonction (du compilateur) qui est chargée de l'émission de la bitmap dans l'assembleur. Elle nous donne une adresse, qui servira de valeur à la chaîne manifeste. On ne peut pas le gérer explicitement ici, puisque cela dépend de l'assembleur.

Un bon langage de programmation cachera la plus des efforts d'implémentation (allocation de la mémoire etc.) au programmeur. Ceci implique que le compilateur devra mettre du code supplémentaire pour certaines opérations.

Dans certains langages la création de structures littérales implique que le compilateur doit faire des allocations mémoire. Dans des conditions pareilles, il faut que le compilateur vide aussi la mémoire quand elle n'est plus nécessaire : il faut du garbage collecting.

### 9.1.3.4 Control Structures

Dans un langage à sémantique fonctionnelle, comment représenter

```
if e_1 then e_2 else e_3
```

Une boucle WHILE

```
test:
  if not (condition)
    goto done
  body
  goto test
done:
```

Traduction de

```
for i := min to max
  do body
```

Proposition 1

```
let i := min
    limit := max
in while i <= limit
  do
    (body; i++)
  end
```

Proposition 2

```
let i := min
    limit := max
in
  if (i > limit)
    goto end
loop:
  body
  if (i >= limit)
    goto end
  i++
  goto loop
end:
```

Les appels de fonctions doivent passer les liens statiques.

### 9.1.4 Function Definition

Doit comprendre plusieurs choses, d'abord le prologue

- Annonce de dbut de fonctions (pour les asm)
- tiquette
- rglage du frame
- sauvegarde des arguments qui chappent (eg, ceux dont on veut l'adresse),
- sauvegarde des registres dont on est responsable.

ensuite le corps de la fonction, enfin, l'pilogue

- mettre la valeur de retour l o il le faut
- restaurer les registres qu'on a protgs
- dsallouer la frame
- le return (ou un jump)
- annonce de la fin de fonction l'intention de l'assembleur.

### 9.1.5 Translation to IR

Sometimes an expression (in the sense of the IR) can be used in another expression, in which case its actual value is needed, and sometimes, it is simply used as an instruction, and its value is thrown away (think of a function call for instance, used only for its side effects and not for its return value).

At other times, this same expression might be used in a condition.

It appears that an expression can be used for three distinct purposes. If we want to produce nice IR, we should take this into account. To this end we will introduce *wrappers* around them : a wrapper for things which are intrinsically expression, a wrapper for instructions, and a wrapper for conditionals (i.e., `cjump`).

#### 9.1.5.1 Translation Samples

The program

```
if 1 < 2 then ()
```

can be translated as

```

SXP
  ESEQ
  SEQ
    CJUMP NE
      CONST 1
      CONST 0
      NAME 19
      NAME 110
    LABEL 19
    MOVE
      TEMP t0
      CONST 2
    JUMP
      NAME 111
    LABEL 110
    MOVE
      TEMP t0
      CONST 0
    JUMP
      NAME 111
    LABEL 111
  SEQ END.
  TEMP t0

```

or, if you work some more

```

SEQ
  CJUMP NE
    CONST 1
    CONST 0
    NAME 19
    NAME 110
  LABEL 19
  SXP
    CONST 2
  JUMP
    NAME 111
  LABEL 110
  SXP
    CONST 0
  JUMP
    NAME 111
  LABEL 111
SEQ END.

```

## 9.2 Normalization

La transformation depuis un langage de haut vers un langage de plus bas niveau est dj bien avance. Pourtant il reste encore trop de concepts de trop haut niveau dans notre RI, on va donc transformer les RI vers quelque chose de plus proche de l'assembleur : on va normaliser les representations.

En fait, la difference essentielle est que l'on doit passer d'une representation en arbre vers une representation lineaire. Pour ce faire, on va s'attaquer aux instructions suivantes :

**ESEQ and SEQ**

il est pratique que les expressions puissent tre values dans n'importe quel ordre, mais ESEQ nous en empche. On fera remonter les ESEQ et SEQ tout en haut, au point de pouvoir mme nous dbarasser des ESEQ.

**CALL** pose le mme problme, et celui de la concurrence sur les registres quand on a des appels de fonctions imbriqués (en clair, on ne veut pas de CALL avec des CALL en paramtres). On exigera des CALL d'tre soit fils de MOVE, soit de EXP.

**CJUMP** doesn't exist in assembly languages: either you jump to an address, or fall to the next instruction.

**9.2.1 Normalization of ESEQ**

Essayer sur les cas suivants

```
ESEQ(s_1, ESEQ(s_2, e))
BINOP(op, ESEQ(s, e_1), e_2)
MEM(ESEQ(s, e_1))
BINOP(op, e_1 ESEQ(s, e_2))
CJUMP(op, e_1 ESEQ(s, e_2), l_1, l_2)
```

Remember that some permutations cannot be made because the order might be important: it doesn't commute. OTOH, it dramatically improves the code if you can recognize some cases which can be simplified.

The end result must be a list of statements, and then a list of expressions.

**9.2.2 Normalization of CALL**

Un des problmes des CALL est le conflit entre les appels sur les registres (en particulier l'occupation des registres servant au passage de paramtres, et celui pour le retour de valeur).

Comment normaliser

```
CALL(foo, foos) + CALL(bar, bars)
```

L'ide la plus simple c'est de rcire

```
CALL(f, args)
```

en

```
ESEQ(MOVE(TEMP(t), CALL(f, args)), TEMP(t))
```

puis laisser la remonte des ESEQ se faire. Ca chifonne un peu l'esprit : on est en train de gnrer beaucoup trop de MOVE, mais on compte sur l'allocation des registres pour amliorer la situation, par la suite.

**9.2.3 Normalization of CJUMP**

Ce sont les CJUMP qui donnent notre RI cette forme d'arbre pas trs minces : on va essayer d'amincir les arbres. Dans le cas des CJUMP(\cdots, l\_f), on va essayer de mettre le lavel l\_f juste derrire.

On sent bien le type d'entites qu'on va manipuler : des blocs.

Un bloc commence par un LABEL, termine par un JUMP ou CJUMP, et n'a aucun des trois dedans.

Une fois les blocs dfinis, on peut les mettre dans n'importe quel ordre dans la mmoire, puisqu'ils ont tous des tiquettes et des sauts ! On va en particulier pouvoir choisir l'ordre qui nous arrange. Comme on s'intresse la vitesse, on cherche viter les instructions inutiles, c'est--dire qu'on va essayer de limiter au maximum les sauts, c'est--dire qu'on va essayer au maximum de mettre derrire les sauts les tiquettes qui leur correspondent. Alors le saut sera inutile.

About CJUMP, since in assembly the ‘else’ part means to fall through, when possible we will move the ‘else’ branch right afterwards. Sometimes it will be advantageous to negate the test, and sometimes another jump might be needed.

Quant aux sauts conditionnels, on essaie de mettre la branche d’chec juse derriere eux.

The consecutive execution of several blocks is called a *trace*. Note that a bloc is defined statically, and a trace, dynamically.

### 9.2.4 Finalization

Il reste garantir la linarite du resultat.

- si un saut conditionnel est suivi de son tiquette succs, inverser le test,
- si suivi ni de succs, ni d’chec, introduire une nouvelle tiquette, et la brancher l o il faut.

A good choice of traces can improve the performances of the compiled code. For instance, here are there different coverings of the following basic blocks:

```

LABEL(prologue)
/* Some prologue. */
JUMP(test)

LABEL(test)
CJUMP(i <= N, body, done)

LABEL(body)
/* Work. */
JUMP(test)

LABEL(done)
LABEL(prologue)
/* Prologue */
JUMP(NAME(test))
LABEL(test)
CJUMP(>, i, N, done, body)
LABEL(body)
/* Body */
JUMP(NAME(test))
LABEL(done)

LABEL(prologue)
/* Prologue */
JUMP(NAME(test))
LABEL(test)
CJUMP(<=, i, N, body, done)
LABEL(done)
/* Epilogue */
LABEL(body)
/* Body */
JUMP(NAME(test))
LABEL(done)

LABEL(prologue)
/* Prologue */
JUMP(NAME(test))
LABEL(body)

```

```
/* Body */  
JUMP(NAME(test))  
LABEL(test)  
CJUMP(>, i, N, done, body)  
LABEL(done)
```

## 10 Instruction selection

We want to have asm from our intermediate representation.

### 10.1 Types des microprocesseurs

CISC processors characteristics include:

- 6, 8, 16 registers, some for pointers, others for int computation
- arithmetic in memory can be processed
- two adresses code
- many possible effects (eg, self-incrementation)

RISC processors characteristics include:

- 32 polyvalent registers
- arithmetic only available on registers
- 3 adresses code
- LOAD and STORE are relative to a register ( $M[r + \text{const}]$ )
- only one effect or result by instruction

We will generate asm RISC related. CISC will have to be adapted by adding supplementary instructions. For instance, our intermediate language uses 3 adresses, production rules will eventually needed for the production of several instruction implying several registers. We want to minimize MOVE utilization, so we need a good register allocation.

Make a correlation between asm instructions and intermediate representation trees modeling it. For example with RISC, STORE which effect is  $M[r.j + c] := r.i$  covers 4 tree patterns ( $\$r_0$  register always contain 0, so there is also the case  $M[c] :=$ ). For some other, we use commutativity, so we understand there is more trees, than significations...

Search for those intructions pattern.(three adressed)

ADD, MUL, SUB, DIV, ADDI, SUBI (const subtraction) LOAD ( $\$r.i := M[r.j + c]$ ), STORE add MOVEM ( $M[r.i] := M[r.j]$ ).

think about the intermediate representation of:

$a[i] := x$

Selecting instruction is of course an combinational optimisaton problem, hence optimal words and optimum have an obvious meaning here. For the proposed solutions, wich are optimum, consider having a cost of 1 for every instructions but MOVEM : we suppose 1, or more.

### 10.2 Slection d'instructions

We don't take care of pipe lines.

#### 10.2.1 Maximal Munch

Start for the root, find the largest tile that can cover it. Emit the assembly instruction which corresponds, and start again on the sub trees.

Note that the instruction are emitted in reversed order.

Try it on the representation of

$a[i] := x$

where  $a$  and  $x$  are in the frame, and  $i$  is in an register.

### 10.2.2 Dynamic Programming

Dynamic programming is a method which can be used when an optimum is always composed of optima.

We start from the bottom, each time we keep the less costing solutions.

Try on  $M[1 + 2]$

That can become very hard to implement. We only want to declare knowledge and let a program make the reflexion. This knowledge can be expressed by an LRR(1) grammar, we should then use generators of code generators.

Though, with RISC processors, it's difficult to find differences between optimum and optimal, therefore we don't need to take care of such complicated things.

### 10.3 Implementation for Tiger

We choose to implement instruction selection before register allocation. Nevertheless instruction selection depends upon the target, while register allocation does not, so we need to find a means to represent target dependent assembly language instruction in a target independent method!

The proposed answer is dead simple: only consider the string representation of these instructions.

But since the register allocation phase will need to know where the register (temporaries) are defined and used, since we also need to trace the labels, we enrich our strings with a list of the temporaries which are defined and used by each instruction.

Finally, the string will not directly hard code the temporaries' names, but simply symbols which denotes either the nth defined or used temporary.

We end up with something like three different classes in the name space 'assem': 'Oper', 'Move', and 'Label'. The class 'Oper' is very general, since it includes the jumps too.

```
Oper (string asm, list<Temp*> dst, list<Temp*> src, list<Label*> jmps)
Move (string asm, Temp &dst, Temp &src)
Label (string asm, Label &label)
```

We have introduced 'Move' because our next task is to get rid of most of these.

For instance, the tree '+ temp1 temp2' becomes

```
Oper ("add 'd0 , 's1, 's2",
      (new temp::Temp ()), (temp1, temp2), ())
```

As you notice, the tiling algorithms must keep us informed of the temporaries they create at each joint of the tree (here the additional 'Temp').

How do you think 'call' should be represented? What of the caller saved registers?

# Index

List of the words used in this document.

## A

Activation record ..... 45

## B

Block Structure ..... 41

## C

Callee-save register ..... 46

Caller-save register ..... 46

## E

Escape ..... 41

## F

Frame ..... 45

## G

Grammar ..... 5

## P

proto-word ..... 5

## S

Stack frame ..... 45

symbol ..... 35

symbol table ..... 43

## T

type checking ..... 43

