**Semantic Actions in a Compiler**

**Parser as Recognizer**

In the theory of formal languages, a parser is simply an algorithm with:

**INPUT:**

1. context-free grammar $G$with a start symbol $S$
2. a word $w$(sequence of terminal symbols)

**OUTPUT:**

1. yes, if $S \Rightarrow^* w$in the grammar $G$
2. no, otherwise

**Example of calculator language:**

Grammar:

term ::= factor ("+" factor)\*

factor ::= INTLITERAL | IDENTIFIER

Recognizer:

[def](http://scala-lang.org) parseTerm : Unit = {

parseFactor

[while](http://scala-lang.org) (lex.token == PLUS) {

lex.next

parseFactor

}

}

[def](http://scala-lang.org) parseFactor : Unit = {

[if](http://scala-lang.org) (lex.current == INTLITERAL) {

lex.next

} [else](http://scala-lang.org) [if](http://scala-lang.org) (lex.current == IDENTIFIER) {

lex.next

} [else](http://scala-lang.org) {

syntaxError("Expected identifier or integer literal")

}

}

**Parser as a Compiler Phase**

In compilers we are interested in generating code, and not just saying whether the program is syntactically correct.

To do so, we extend the “yes/no” parser with **semantic actions**, which do something with the program content, using the information on \*how\* the program was parsed.

**Pretty Printing**

Automatically format program according to some conventions (indentation, etc)

[def](http://scala-lang.org) parseTerm : Unit = {

parseFactor

[while](http://scala-lang.org) (lex.token == PLUS) {

lex.next

print("\n + ") // put each factor in new line

parseFactor

}

}

[def](http://scala-lang.org) parseFactor : Unit = {

[if](http://scala-lang.org) (lex.current == INTLITERAL) {

[val](http://scala-lang.org) v = lex.getIntValue

printInt(v) // will print all constants in decimal, even if they were parsed as hex

lex.next

} [else](http://scala-lang.org) [if](http://scala-lang.org) (lex.current == IDENTIFIER) {

[val](http://scala-lang.org) name = lex.getIdent

print(name)

lex.next

} [else](http://scala-lang.org) {

syntaxError("Expected identifier or integer literal")

}

}

**Interpreting (Evaluating) Program**

If we built an interpreter for very simple language and did interpretation while parsing (not very efficient if language has loops)

[def](http://scala-lang.org) parseTerm : Int = {

[var](http://scala-lang.org) sum = parseFactor

[while](http://scala-lang.org) (lex.token == PLUS) {

lex.next

sum = sum + parseFactor

}

sum

}

[def](http://scala-lang.org) parseFactor : Int = {

[if](http://scala-lang.org) (lex.current == INTLITERAL) {

[val](http://scala-lang.org) res = lex.getIntValue

lex.next

res

} [else](http://scala-lang.org) [if](http://scala-lang.org) (lex.current == IDENTIFIER) {

[val](http://scala-lang.org) name = lex.getIdent

lex.next

lookup(name)

} [else](http://scala-lang.org) {

syntaxError("Expected identifier or integer literal")

}

}

**Emitting Code**

In simple compilers, semantic actions can already emit the code.

[def](http://scala-lang.org) parseTerm : List[Instruction] = {

[var](http://scala-lang.org) instrs = parseFactor

[while](http://scala-lang.org) (lex.token == PLUS) {

lex.next

instrs = instrs ::: parseFactor :: List(AddInstruction)

}

instrs

}

[def](http://scala-lang.org) parseFactor : List[Instruction] = {

[if](http://scala-lang.org) (lex.current == INTLITERAL) {

[val](http://scala-lang.org) v = lex.getIntValue

lex.next

List(PushConstantInstruction(v))

} [else](http://scala-lang.org) [if](http://scala-lang.org) (lex.current == IDENTIFIER) {

[val](http://scala-lang.org) name = lex.getIdent

lex.next

List(LoadInstruction(lookupAddress(name)))

} [else](http://scala-lang.org) {

syntaxError("Expected identifier or integer literal")

}

}

**Building Abstract Syntax Tree**

For all more complex actions, semantic actions build **abstract syntax tree**, a representation of the program that has all relevant information about how the program should execution (but not necessarily how the program is written e.g. where the comments were, etc.).

Here is a tiny syntax tree:

[sealed](http://scala-lang.org) [abstract](http://scala-lang.org) [class](http://scala-lang.org) Expression

[case](http://scala-lang.org) [class](http://scala-lang.org) IntegerLiteral(v : Int) [extends](http://scala-lang.org) Expressioon

[case](http://scala-lang.org) [class](http://scala-lang.org) Variable(id : String) [extends](http://scala-lang.org) Expression

[case](http://scala-lang.org) [class](http://scala-lang.org) Plus(e1 : Expression, e2 : Expression) [extends](http://scala-lang.org) Expression

Here is the version of the parser that builds the syntax tree:

[def](http://scala-lang.org) parseTerm : Expression = {

[var](http://scala-lang.org) e = parseFactor

[while](http://scala-lang.org) (lex.token == PLUS) {

lex.next

e = Plus(e, parseFactor)

}

e

}

[def](http://scala-lang.org) parseFactor : Expression = {

[if](http://scala-lang.org) (lex.current == INTLITERAL) {

[val](http://scala-lang.org) v = lex.getIntValue

lex.next

IntegerLiteral(v)

} [else](http://scala-lang.org) [if](http://scala-lang.org) (lex.current == IDENTIFIER) {

[val](http://scala-lang.org) name = lex.getIdent

lex.next

Variable(name)

} [else](http://scala-lang.org) {

syntaxError("Expected identifier or integer literal")

}

}

When we run the code above on

xx + 42 + yy

we obtain the tree:

Plus(Plus("xx", IntegerLiteral(42)),

Variable("yy"))