G53CMP Compilers: Coursework 1

20/10/2014

Ryan Shaw
rxs62u

October 20, 2014

A Task I.1

A.1 Grammar extension

First all the grammars are extended to include the new loop construct

A.1.1 Lexical Syntax

Keyword → `begin` | `const` | `do` | `else` | `end` | `if` | `in`
| `let` | `then` | `var` | `while` | `repeat` | `until`

A.1.2 Context-Free Syntax

Command → *VarExpression* `:=` *Expression*
| *VarExpression* `(` *Expressions* `)`
| `if` *Expression* `then` *Command*
`else` *Command*
| `while` *Expression* `do` *Command*
| `let` *Declarations* `in` *Command*
| `begin` *Commands* `end`
| `repeat` *Command* `until` *Expression*

A.1.3 Abstract Syntax

Program	→	*Command*	Program
Command	→	*Expression* `:=` *Expression*	CmdAssign
	|	*Expression* `(` *Expression** `)`	CmdCall
	|	`begin` *Command** `end`	CmdSeq
	|	`if` *Expression* `then` *Command* `else` *Command*	CmdIf
	|	`while` *Expression* `do` *Command*	CmdWhile
	|	`let` *Declaration** `in` *Command*	CmdLet
	|	`repeat` *Command* `until` *Expression*	CmdRepeat

Non-terminals are typeset in italics, like *this*. Terminals are typeset in typewriter font, like `this`. Terminals whose spelling (the concrete character sequence) is different from what is shown in the grammar are typeset in italics and underlined, like *Identifier* and *IntegerLiteral*. Their spelling is defined by the lexical grammar (where they are non-terminals!).

Program → *Command*

Commands → *Command*
| *Command* `;` *Commands*

Command → *VarExpression* `:=` *Expression*
| *VarExpression* `(` *Expressions* `)`
| `if` *Expression* `then` *Command*
`else` *Command*
| `while` *Expression* `do` *Command*
| `let` *Declarations* `in` *Command*
| `begin` *Commands* `end`

Expressions → *Expression*
| *Expression* `,` *Expressions*

Expression → *PrimaryExpression*
| *Expression BinaryOperator Expression*

PrimaryExpression → *IntegerLiteral*
| *VarExpression*
| *UnaryOperator PrimaryExpression*
| `(` *Expression* `)`

VarExpression → *Identifier*

BinaryOperator → `^` | `*` | `/` | `+` | `-` | `<` | `<=` | `==` | `!=` | `>=` | `>` | `&&` | `||`

UnaryOperator → `-` | `!`

Declarations → *Declaration*
| *Declaration* `;` *Declarations*

Declaration → `const` *Identifier* `:` *TypeDenoter* `=` *Expression*
| `var` *Identifier* `:` *TypeDenoter*
| `var` *Identifier* `:` *TypeDenoter* `:=` *Expression*

TypeDenoter → *Identifier*

Note that the productions for *Expression* makes the grammar as stated above ambiguous. Operator precedence and associativity for the *binary* op-

erators as defined in the following table is used to disambiguate:

`Operator`	Precedence	Associativity
`^`	1	right
`* /`	2	left
`+ -`	3	left
`< <= == != >= >`	4	non
`&&`	5	left
`||`	6	left

A precedence level of 1 means the highest precedence, 2 means second highest, and so on.

Also note that the syntactic category *Declaration* includes *definition* of constants, as not only the type is given, but also the value of the constant (through an expression), and that a variable declaration includes an optional initialization.

A.2 MiniTriangle Abstract Syntax

This is the MiniTriangle abstract syntax. It captures the tree structure of MiniTriangle programs as concisely as possible. It is used as the basis for designing the datatypes for representing MiniTriangle programs. For example, note that there is only one non-terminal for expressions as opposed to three in the grammar for the concrete syntax. The extra non-terminals (along with a specification of binary operator associativity and precedence) are needed to make the concrete syntax *unambiguous*, which is necessary for parsing. But once a program has been successfully parsed, its structure has been determined, and ambiguity is no longer an issue. Another difference is that general function application has been introduced as one expression form. This replaces both concrete unary operator application and concrete (infix) binary operator application, as such operators *are* functions of one and two arguments, respectively. (The concrete syntax of MiniTriangle currently does not include general function application, but that could easily be added.) As a consequence, a single "variable" terminal *Name* also replaces both *Identifier* and *Operator*; i.e., $\underline{\textit{Identifier}} \subseteq \underline{\textit{Name}}$ and $\underline{\textit{Operator}} \subseteq \underline{\textit{Name}}$.

The rightmost column gives the node labels for drawing abstract syntax trees. (They correspond to the names of the data constructors of the of the abstract syntax datatypes in the compiler.) Note that some elements of concrete syntax, such as keywords, do occur in the productions. They are there to make the connection between the concrete and abstract syntax clear, and to provide an *alternative* textual representation for the abstract syntax (e.g. for use in typing rules). However, these fragments of concrete syntax are

omitted when drawing abstract syntax trees, as they are implied by the node labels and thus superfluous. Also note that some of the productions make use of the EBNF *-notation for sequences. When drawing an abstract syntax tree, that means that the corresponding nodes will have a varying number of children.

Program	→	*Command*	Program
Command	→	*Expression* `:=` *Expression*	CmdAssign
	|	*Expression* `(` *Expression** `)`	CmdCall
	|	`begin` *Command** `end`	CmdSeq
	|	`if` *Expression* `then` *Command* `else` *Command*	CmdIf
	|	`while` *Expression* `do` *Command*	CmdWhile
	|	`let` *Declaration** `in` *Command*	CmdLet
Expression	→	*IntegerLiteral*	ExpLitInt
	|	*Name*	ExpVar
	|	*Expression* `(` *Expression** `)`	ExpApp
Declaration	→	`const` *Name* `:` *TypeDenoter* `=` *Expression*	DeclConst
	|	`var` *Name* `:` *TypeDenoter* (`:=` *Expression* | ϵ)	DeclVar
TypeDenoter	→	*Name*	TDBaseType