LFP: A Logic for Linguistic Descriptions and an Analysis of its Complexity

William C. Rounds

Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109

We investigate the weak expressive power of a notation using first-order logic, augmented with a facility for recursion, to give linguistic descriptions. The notation is precise and easy to read, using ordinary conventions of logic. Two versions of the notation are presented. One, called CLFP, speaks about strings and concatenation, and generates the class EXPTIME of languages accepted by Turing machines in time 2^{cn} for some $c > 0$. The other, called ILFP, speaks about integer positions in strings, and generates the class PTIME of languages recognizable in polynomial time. An application is given, showing how to code Head Grammars in ILFP, showing why these grammars generate only polynomial time languages.

1 First-order Logic as a Tool for Syntactic Description

In this paper we investigate the properties of a new notation for specifying syntax for natural languages. It is based on the simple idea that first-order logic, though inadequate as a semantics for natural language, is quite adequate to express relations between syntactic constituents. This is the insight behind definite clause grammars (DCGs) (Pereira and Warren 1980) and, in fact, our notation is in some ways a generalization of that notation. However, we have tried to keep our formalism as much as possible like that of standard textbook first-order logic. There are actually two versions of our notation. The first works with strings of symbols and uses concatenation as a primitive operation. The second works with integers and takes the standard arithmetic operations as primitive. These integers can be regarded as indexing positions of morphemes in a sentence, but the sentence itself is not explicitly referenced. Both versions allow the recursive definition of predicates over strings and integers. This capacity for recursive definition is what gives our grammars their generative ability, and our notation has this feature in common with DCGs. However, we liberate DCGs from the Horn clause format, and we do not base the semantics of our notation on the semantics for Prolog or for logic programs. We hope that making the logic more familiar and readable will encourage more people to use logic as a means for specifying desired syntactic relations between sentential constituents in grammars. Anyone knowing the standard conventions of first-order logic should be able to read or to specify a grammar in our notation.

We also provide a precise semantics for our two notations. This involves using the least-fixed-point operator from denotational semantics for programming languages to explain the recursive definition of predicates. It involves as well using restricted universal and existential quantification to restrict the class of definable predicates (sets of strings). We prove a complexity theoretic characterization for both grammar formalisms: (1) the formalism using strings and concatenation defines exactly the class EXPTIME of formal languages recognizable by deterministic Turing machines within time $T(n) = 2^{cn}$ for some positive c; and (2) the formalism using integers defines exactly the class PTIME of languages recognizable in time $T(n) = n^k$ for some integer k.

As an application of the second notation we sketch a natural way to write Head Grammars (Pollard 1984). Because these grammars can be expressed in this way, we immediately obtain the result that head languages can be recognized in polynomial time. We even obtain an estimate of the degree of the polynomial that is required, derived directly from the form of the grammatical description. Unfortunately, the estimated degree is at least twice as large as is actually necessary if one uses the algorithm of Pollard (1984), or Vija-

0362-613X/88/01001-9$03.00

yashanker and Joshi (1985). We conjecture that in fact, this factor of (2) can be removed.

Our complexity theoretic characterizations are versions of theorems already appearing in the literature. Shapiro (1984) characterizes the class of languages definable by logic programs with a linear space restriction as the class EXPTIME. The proof of our first theorem is very much like his. Our second theorem characterizing PTIME can be viewed as a specialization of the results of Chandra and Harel (1982), Immerman (1982), and Vardi (1982), who show that the class of sets of finite mathematical structures definable by formulas of first-order logic augmented with a least-fixed-point operator is just the class of sets of structures recognizable in polynomial time. We prove both of our results in the same way, and thus show how these apparently unconnected theorems are related. The proof uses the notion of alternating Turing machines, and thereby explains the significance of this idea for the science of formal linguistics.

We should also note that our notation will not find immediate application in current linguistic theory, because it does not allow structural descriptions to be described. We are in the process of extending and modifying the notation for this purpose. However, we think it is important to explicate the properties of the individual operations used in building strings and structures. Our first attempt is therefore to understand how concatenation of strings can be expressed in a restricted logic. We can then consider other predicates or functions on both strings and treelike structures in the same uniform way.

2 CLFP Grammars: Grammars Based on Concatenation Theory

2.1 Syntax of CLFP

We present a standard version of the first-order theory of concatenation, augmented with the least-fixed-point operator. Before proceeding with the formal description, we give an example to illustrate the scheme we have in mind. Consider the following context-free fragment, adapted directly from Johnson (1985).

$S \rightarrow NP\ VP$

$NP \rightarrow Det\ Noun$

$VP \rightarrow V\ NP$

$Det \rightarrow NP[+Gen] \mid the$

Here is the corresponding CLFP fragment:

$$S(x) \Leftrightarrow \exists yz.NP[-Gen](y) \wedge VP(z) \wedge x = yz;$$

$$NP[case](x) \Leftrightarrow \exists yz.Det(y) \wedge Noun[case](z) \wedge x = yz;$$

$$VP(x) \Leftrightarrow \exists yz.V(y) \wedge NP[-Gen](z) \wedge x = yz;$$

$$Det(x) \Leftrightarrow NP[+Gen](x) \vee x = \text{the}.$$

In this formulation, x,y, and z range over strings of symbols (morphemes) and NP,VP, etc. are predicates over strings. The second clause is here an abbreviation for two clauses, where *case* can take two values, namely $+Gen$ and $-Gen$. At present we do not treat the problem of calculating complex feature structures, but there seems to be no reason that the notation cannot be suitably extended.

This example illustrates the most complex case of a CLFP formula. It is a **recursion scheme**, which assigns to predicate variables, S,NP, etc. certain formulas (the right-hand sides of the corresponding clauses in the definition). The whole scheme is understood as the simultaneous recursive definition of the predicate variables in the left sides of the definition. To handle the fact that string variables occur on the left-hand side of each clause, we will understand each clause as a function assigning both the formula on its right and the set of individual variables mentioned on the left to the given predicate symbol.

We now proceed with the formal definition of CLFP. Let **Ivar** be a set $\{x_0,x_1, \ldots\}$ of individual variables ranging over strings. Let Σ be a finite set of terminal symbols. These are the constants of our theory. Λ is another constant denoting the null string. Terms are built from variables and constants using the binary operation of concatenation. We also require a set **Pvar** of predicate variables, listed as the set $\{P_1,P_2, \ldots\}$. Each predicate variable P is equipped with an arity $ar(P)$, indicating the number of individual arguments that a relation assigned to this variable will have. (The example CLFP scheme given above employs only unary predicate variables S,NP,VP, and Det.) The set of CLFP formulas is given by the following inductive clauses.

1. If $P \in$ Pvar and $(x_1, \ldots,x_n)$ is a sequence of Ivar with length $n = ar(P)$ then $P(x_1, \ldots,x_n)$ is in CLFP;
2. If t_1 and t_2 are terms, then $t_1 = t_2$ is in CLFP;
3. If $x \in$ Ivar and ϕ is in CLFP then $\exists x\phi$ and $\forall x\phi$ are in CLFP;
4. The usual Boolean combinations of CLFP formulas are in CLFP.
5. This clause requires more definitions. Let $\mathcal{R}$ be a finite nonempty subset of Pvar with a distinguished element $S \in \mathcal{R}$. Let $\Phi : \mathcal{R} \rightarrow \mathcal{P}(Ivar) \times CLFP$. ($\Phi(R)$ is going to be the defining clause for the predicate R.) If $\Phi(R) = (X,\phi)$, then we define $B\Phi(R) = X$, and $C\Phi(R) = \phi$. We require that $|B\Phi(R)| = ar(R)$ and thus be a finite set of individual variables. Now we say that the whole map Φ is a **recursion scheme** iff each $P \in \mathcal{R}$ occurs only positively in $\Phi(R)$ for any $R \in \mathcal{R}$; that is, within the scope of an even number of negation signs. Finally, condition 5 states that if Φ is a recursion scheme, with distinguished variable S, then $\mu S\Phi$ (the least fixed point of Φ) is in CLFP.

Example 1. Consider the following scheme, which defines a regular language.

$$S(x) \Leftrightarrow \exists y((x = ay \wedge (S(y)) \vee (x = by \wedge T(y))) \vee x = a$$

$$T(v) \Leftrightarrow \exists w(v = cw \wedge S(w)).$$

In this example, $\mathcal{R} = \{S,T\}$, $B\Phi(S) = \{x\}$, and

$$C\Phi(S) = \exists y((x = ay \wedge (S(y)) \vee (x = by \wedge T(y))) \vee x = a.$$

Similarly, $B\Phi(T) = \{v\}$, and $C\Phi(T)$ is the second formula in the scheme.

In the example, we have written our recursion scheme in a conventional style to emphasize its direct connection to the usual grammatical presentations. Thus the variable x is bound by the left-hand side of (1), so this clause has been written with $S(x)$ on the left to make this fact apparent. Also, the use of the $\Leftrightarrow$ sign is conventional in writing out Φ. In our example, the first clause is taken as defining the distinguished predicate S of our scheme. Finally, there are no occurrences of free predicate variables in this example, but there are in our first example (e.g., *noun*).

The usual rules for calculating free individual variables apply; if $Fvar(\phi)$ is the set of free variables of ϕ, then $FIvar(P(x_1, \ldots, x_n)) = \{x_1, \ldots, x_n\}$. The quantifier and Boolean cases are handled as in standard text presentations. However, if Φ is a recursion scheme then $FIvar(\mu S\Phi)$ will be calculated as follows. For each $R \in \mathcal{R}$, find $Fvar(C\Phi(R))$. Remove from this set any variables in $B\Phi(R)$. The union of the resulting sets for each $R \in \mathcal{R}$ is defined to be the set $FIvar(\mu S\Phi)$.

The rules for free predicate variables are a bit simpler. In the atomic formula $P(x_1, \ldots, x_n)$, P is a free predicate variable. In a recursion scheme Φ with domain $\mathcal{R}$, the set $FPvar(\mu S\Phi))$ is the union of the sets $FPvar(\mu S\Phi(R)))$, minus the set $\mathcal{R}$.

A final remark on notation: we will use the notation $\phi(t_1, \ldots, t_n)$ to stand for the formula

$$\exists x_1 \ldots \exists x_n(\phi(x_1, \ldots, x_n) \wedge x_1 = t_1 \wedge \ldots \wedge x_n = t_n)$$

where the t_i are terms, and the x_i are individual variables not appearing in any ts. This will not affect our complexity results in any way.

2.2 SEMANTICS FOR CLFP

We borrow some notation from the Oxford school of denotational semantics to help us explain the meaning of our logic grammars. If X and Y are sets, then $[X \to Y]$ is the set of functions from X to Y. Let $A = [\text{Ivar} \to \Sigma^*]$ be the set of **assignments** of values to individual variables. We wish to define when a given assignment, say α, satisfies a given CLFP formula ϕ. This will depend on the meaning assigned to the free predicate variables in ϕ, so we need to consider such assignments. Let PA be the set of maps ρ from Pvar to the class of relations on Σ^* such that the arity of $\rho(P)$ is $ar(P)$. We are now ready to define for each formula ϕ and predicate assignment ρ, the set $\mathcal{M}[\![\phi]\!]\rho \subseteq A$ of individual assignments satisfying ϕ with respect to ρ.

1. $\mathcal{M}[\![P(x_1, \ldots, x_n)]\!]\rho = \{\alpha \mid \langle\alpha(x_1), \ldots, \alpha(x_n)\rangle \in \rho(P)\}$;
2. $\mathcal{M}[\![t_1 = t_2]\!]\rho = \{\alpha \mid t_1\alpha = t_2\alpha\}$, where $t\alpha$ is the evaluation of t with variables assigned values by α;
3. $\mathcal{M}[\![\exists x\phi]\!]\rho = \{\alpha \mid \exists u \in \Sigma^* : \alpha(x/u) \in \mathcal{M}[\![\phi]\!]\rho\}$, and similarly for universal quantification;
4. $\mathcal{M}[\![\phi \vee \psi]\!]\rho = \mathcal{M}[\![\phi]\!]\rho \cup \mathcal{M}[\![\psi]\!]\rho$, and similarly for other Boolean connectives.
5. $\mathcal{M}[\![\mu S\Phi]\!]\rho = \{\alpha \mid (\exists k)(\alpha \in \mathcal{M}[\![C\Phi^k(S)]\!]\rho)\}$

where, for each k, Φ^k is a recursion scheme with the same domain $\mathcal{R}$ as Φ, and is defined as follows by induction on k. First, we stipulate that for each $P \in \mathcal{R}$, the set $B\Phi^r(P) = B\Phi(P)$. Then we set

$$C\Phi^0(P) = C\Phi(P)[R \leftarrow FALSE : R \in \mathcal{R}];$$

$$C\Phi^{k+1}(P) = C\Phi(P)[R \leftarrow C\Phi^k(R) : R \in \mathcal{R}]$$

where the notation $\psi[R \leftarrow \theta(R) : R \in \mathcal{R}]$ denotes the simultaneous replacement of atomic subformulas $R(w_1, \ldots, w_k)$ in ψ (where R is a free occurrence) by the formula $\theta(R)(w_1, \ldots, w_k)$, in such a way that no free occurrences of other variables in $\theta(R)$ are captured by a quantifier or a μ-operator in ψ. (We may always change the bound variables of ψ first, to accomplish this.)

This definition appears much more clumsy than it really is, and we continue our example to illustrate it. Refer to the example of a regular grammar in the previous section. In the clause for S we are required to substitute the formula FALSE for occurrences of both S and T. This gives, after simplification,

$$C\Phi^0(S)(x) = (x = a).$$

Similarly, substitution of FALSE into the clause for T gives $\Phi^0(T)(v) = \text{FALSE}$. Now substitution of these new formulae for S and T into Φ gives (after simplification):

$$C\Phi^1(S)(x) = \exists y(x = ay \wedge x = a) \vee x = a;$$

$$C\Phi^1(T)(v) = \exists w(v = cw \wedge w = a).$$

It is easy to see that continuing this process will simulate all possible derivations in the grammar, and also that it explains the meaning of the scheme Φ in terms of the meaning of subformulas.

Some remarks are in order to explain why we use the term "least-fixed-point", and to explain why, in a recursion scheme, all occurrences of recursively called predicates are required to be positive. Let $\Phi : \mathcal{R} \to$ CLFP be a recursion scheme. Define the map $T[\![\Phi]\!] : PA \to PA$ as follows. If $R \in \mathcal{R}$, then

$$\langle u_1, \ldots, u_n\rangle \in T[\![\Phi]\!]\rho(R) \Leftrightarrow (\exists \alpha \in \mathcal{M}[\![\Phi(R)]\!]\rho)(\alpha(x_i) = u_i)$$

where $\langle x_1, \ldots, x_n\rangle$ is the sequence of variables in $B\Phi(R)$, listed in increasing order of subscripts. If $R \notin \mathcal{R}$, then $T[\![\Phi]\!]\rho(R) = \rho(R)$. Next, let

$$T[\![\mu R\Phi]\!]\rho = \bigcup_{k \geq 1} T[\![\Phi]\!]^{(k)}(\rho[R \leftarrow \emptyset : R \in \mathcal{R}])$$

where unions are coordinatewise, $F^{(k)}$ is the k-th iterate of F, and $\rho[R \leftarrow \emptyset : R \in \mathcal{R}]$ is ρ with the empty relation assigned to each predicate variable in $\mathcal{R}$. This formula is just the familiar least-fixed-point formula $\sqcup_{k \geq 1} F^{(k)}(\perp)$ from denotational semantics. Then we can check that $\mathcal{T}[\![\mu S \Phi]\!]\rho$ is in PA, and is the least fixed point of the continuous map $\mathcal{T}[\![\Phi]\!]$. It is then possible to prove that

$$\mathcal{M}[\![\mu S \Phi]\!]\rho = (\mathcal{T}[\![\mu S \Phi]\!]\rho)(S)$$

where S is the distinguished predicate variable in $\mathcal{R}$.

If we had no conditions on negative formulas in recursion schemes, then we could entertain schemes like

$$S(x) \Leftrightarrow \neg S(x)$$

which, although they would receive an interpretation in our first definition, would give a T which was not continuous, or even monotonic. We therefore exclude such cases for reasons of smoothness.

Next we come to the definition of the language or relation denoted by a formula. A k-ary relation P on Σ^* is said to be definable in CLFP iff there is a CLFP formula ϕ with no free predicate variables such that $\langle u_1, \ldots, u_k \rangle \in P \Leftrightarrow \exists \alpha \in \mathcal{M}[\![\phi]\!]: \langle \alpha(x_1), \ldots, \alpha(x_k) \rangle = \langle u_1, \ldots, u_k \rangle$, where $x_1, \ldots, x_k$ is the list of free variables in ϕ arranged in increasing order of subscript. (Notice that the parameter ρ has been omitted since there are no free predicate variables in ϕ.)

So far, we have not restricted quantification in our formulas, and every r.e. predicate is definable. We need to add one other parameter to the definition of $\mathcal{M}$, which will limit the range of quantification. This will be an integer n, which will be the length of an input sentence to be recognized. The occurrences of the formula $\mathcal{M}[\![\phi]\!]\rho$ will thus be changed everywhere in the above clauses to $\mathcal{M}[\![\phi]\!]\rho n$. The only change in the substance of the clauses is in the rule for existential and universal quantification.

$$\mathcal{M}[\![\exists x \phi]\!]\rho n = \{\alpha \mid \exists u \in \Sigma^* : |u| \leq n \wedge \alpha(x/u) \in \mathcal{M}[\![\phi]\!]\rho n\}.$$

A predicate P is said to be boundedly definable iff for some ϕ:

$$\langle u_1, \ldots, u_k \rangle \in P \Leftrightarrow \exists \alpha \in \mathcal{M}[\![\phi]\!]n : \langle \alpha(x_1), \ldots, \alpha(x_k) \rangle = \langle u_1, \ldots, u_k \rangle$$

where $n = \max(|u_i|)$. (To abbreviate the right-hand condition, we write $\langle u_1, \ldots, u_k \rangle \models \phi$). Our first theorem can now be stated.

Theorem 1. A language (unary predicate) is boundedly definable in CLFP iff it is in EXPTIME.

We defer the proof to the next section.

3 EXPTIME AND CLFP

3.1 ALTERNATION

Before proving Theorem 1, we need to discuss the method of proof both for this result and for the Integer LFP characterization of PTIME in the next section. This material is repeated from the fundamental article of Chandra, Kozen, and Stockmeyer (1981). Their paper should be consulted for the full details of what we state here.

An alternating Turing machine can be regarded as a Turing machine with unbounded parallelism. In a given state, and with given tape contents, the machine can spawn a finite number of successor configurations according to its transition rules. These configurations are thought of as separate processes, each of which runs to completion in the same way. A completed process is one which is in a special accepting state with no successors. The results of the spawned processes are reported back to the parent, which combines the results to pass on to its own parent, and so forth. How the parent does this depends on the state of the finite control. These states are classified as being either **existential** (OR), **universal** (AND), **negating** (NOT), or accepting. If the parent is in an existential state, it reports back the logical OR of the results of its offspring. If it is in a universal state, it reports back the logical AND; if the state is negating, the parent reports the negation of its one offspring. An accepting state generates a logical 1 (TRUE) to be reported back. Thus a nondeterministic TM can be regarded as an alternating TM with purely existential states.

An alternating TM is defined as a tuple in a standard way. It has a read-only input tape with a head capable of two-way motion. It also has a fixed number of work tapes. The input tape contains a string $u \in \Sigma^*$, while the work tapes can use a tape alphabet Γ. The transition relation δ is defined as for ordinary nondeterministic TMs. The state set is partitioned as described above into universal, existential, negating, and accepting states. The relation δ is constrained so that existential and universal states have at least one successor, negating states have exactly one successor, and accepting states have no successors. A **configuration** is then just a tuple describing the current state, positions of the heads, and tape contents as is familiar. The initial configuration is the one with the machine in its initial state, all the work tapes empty, and the input head at the left end of the input u. The **successor** relation $\vdash$ between configurations is defined again as usual.

To determine whether or not a configuration is accepting, we proceed as follows. Imagine the configurations that succeed the given one arranged in a tree, with the given configuration at the root. At each node, the immediate descendants of the configuration are the successors given by $\vdash$. The tree is truncated at a level determined by the length of the input tape (this truncation is not part of the general definition but will suffice for our results.) The leaf nodes of this tree are labeled with (0) if the configuration at that node is not accepting, and with (1) if the configuration is accepting. The tree is then evaluated according to the description given above. The configuration at the root is accepting iff it is labeled (1) by this method. Thus an input is accepted by

the machine iff the initial configuration with that input is accepting. In our application, it will always suffice to cut off the tree at level 2^{cn}, where n is the length of the input string, and c is a positive constant depending only on the description of the machine.

We say that an alternating TM is $S(n)$ space bounded iff in the above tree, for any initial configuration labeling the root, no auxiliary tape length ever exceeds $S(n)$ where n is the length of the input. We are concerned only with the functions $S(n) = \log n$ and $S(n) = n$ in this paper. We let the class $ASPACE(S(n))$ be the class of languages accepted by space-bounded ATMs in this way. We then have the following theorem (Chandra, Kozen, Stockmeyer 1987):

Lemma 1. If $S(n) \geq \log n$, then

$$ASPACE(S(n)) = \bigcup_{c>0} DTIME(2^{cS(n)})$$

where $DTIME(T(n))$ is the class of languages accepted deterministically by ordinary Turing machines within $T(n)$ steps.

Our problem in the rest of this section is to show how linear space bounded ATMs and CLFP grammars simulate each other. To facilitate the construction of the next section, it is convenient to add one feature to the definition of alternating Turing machines. Let U be the name of a k-ary relation on Σ^*. We allow machines to have **oracle states** of the form $U?(i_1, \ldots, i_k)$, where the i_j are tape numbers. If now the predicate U is interpreted by an actual relation on Σ^*, then when M executes such an instruction, it will accept or reject according to whether the strings on the specified tapes are in the relation U. We will need such states to simulate recursive invocations of recursion schemes. It is not hard to modify the definition of acceptance for ordinary ATMs to that for oracle ATMs. The language or relation accepted by the ATM will now of course be relative to an assignment ρ of relations to the predicate names U.

The next subsections contain our constructions for the CLFP characterizations. Then, in Section 4 we will treat Integer LFP grammars and show how these grammars and logspace bounded ATMs simulate each other. As a consequence of the above lemma, we will then have our main results.

3.2 PROOF OF THEOREM 1

Our first task is to show that if a language L is (boundedly) CLFP-definable, then it can be recognized by a linear space bounded ATM. The idea is simple. Given an input string, our machine will try to execute the logical description of the grammar. Its states will correspond to the logical structure of the CLFP formula. If that formula is, for example, the logical AND of two subformulas, then the part of our machine for that formula will have an AND state. A recursion scheme will be executed with states corresponding to the predicate variables involved in the recursion, and so forth.

To give an explicit construction of an ATM corresponding to a formula ϕ of CLFP we need to be precise about the number of work tapes required. This will be the sum of the number of free individual variables of ϕ, and the number of "declarations" of bound variables in ϕ. A "declaration" is either the occurrence of a universal or existential quantifier in ϕ, or one of the individual variables bound on the left side of a (non-S) clause in a recursion scheme. If that clause defines the predicate R, then the number of variables declared at that point is $ar(R) = |B\Phi(R)|$. We thus define the number $\beta(\phi)$ of declarations of bound variables in ϕ by induction as follows:

1. $\beta(R(x_1, \ldots, x_n)) = 0$;
2. $\beta(t_1 = t_2) = 0$;
3. $\beta(\phi \vee \psi) = \beta(\phi \wedge \psi) = \beta(\phi) + \beta(\psi)$;
4. $\beta(\neg\phi) = \beta(\phi)$;
5. $\beta(\exists x\phi) = \beta(\forall x\phi) = 1 + \beta(\phi)$;
6. $\beta(\mu S\Phi) = \beta(C\Phi(S)) + \Sigma_{R \in \mathcal{R}\setminus\{S\}}(ar(R) + \beta(C\Phi(R)))$.

The number $\gamma(\phi)$ counts the maximum number of tapes needed, and is defined to be $\beta(\phi) + |Fivar(\phi)| + 1$.

We can now state the inductive lemma which allows the construction of ATMs.

Lemma 2. Let ϕ be a CLFP formula, with $|FIvar(\phi)| = k$, and $T:FIvar(\phi) \to \{1, \ldots, k\}$. Let $m = \gamma(\phi)$. Then we may construct an m-tape ATM $M(\phi,T)$ having the following properties: (i) M has oracle states P? for each free predicate variable of ϕ, and (ii) For any $\alpha:FIvar(\phi) \to \Sigma^*$, and any environment ρ, we have the following. Let $n = \max\{|\alpha(x_i)|\}$. Then M with oracle states for the $\rho(P)$, started with $\alpha(x_1)$ on tape $T(x_1)$, . . ., and $\alpha(x_k)$ on tape $T(x_k)$, and the other tapes blank, will accept without ever writing more than n symbols on any tape, if and only if $\langle\alpha(x_1), \ldots, \alpha(x_k)\rangle \in \mathcal{M}[\![\phi]\!]\rho n$.

Proof: This lemma formalizes the intuitive idea, stated above, that to calculate the membership of a string x in the language defined by a recursion scheme, it suffices to execute the scheme recursively. The full proof would use the formal definition of the semantics of ATMs, which themselves are given by least-fixed-point definitions. We have chosen not to give the full proof, because the amount of explanation would be overwhelming relative to the actual content of the proof. Instead we give a reasonably complete account of the inductive construction involved, and illustrate with the regular set example of the previous section.

To start the induction over formulas ϕ, suppose that ϕ is $R(x_1, \ldots, x_k)$. Then we may take M to be a machine with $k = \gamma(\phi)$ tapes, with one oracle state P, and the single instruction $P?(T(x_1), \ldots, T(x_k))$.

If ϕ is $t_1 = t_2$, then we let M be a simple machine evaluating t_1 and t_2, using perhaps an extra tape for bookkeeping. It does a letter-by-letter comparison, so that it never has to copy more than the maximum length of any one tape.

If ϕ is $\neg\psi$, then $M(\phi)$ consists of adding a negating

state before the initial state of $M(\psi)$, and transferring control to that initial state.

If ϕ is $\psi_1 \vee \psi_2$, we construct M_1 and M_2 by inductive hypothesis. Then $M(\phi)$ is constructed by having disjoint instruction sets corresponding to each M_i, prefixed by an OR state which transfers control to either of the two formerly initial states. The free individual variables of the disjunction are those occurring free in either disjunct. Let T be an assignment of tapes to the free variables of the disjunction. Then we construct M_1 with a T_1 such that $T_1(x) = T(x)$, and similarly for M_2, where x is a free individual variable. Otherwise, any tapes referenced in M_1 are distinct from any tapes referenced in M_2. In other words, the machine M has shared storage for the free variables, and private storage for variables bound in either disjunct. The oracle states in the two pieces of code are not made disjoint, however, because a predicate variable is free in the disjunction iff it is free in either disjunct. It is clear that the number of tapes of the $\psi_1 \vee \psi_2$ is just $\gamma(\psi_1 \vee \psi_2)$. For the case of $\phi = \psi_1 \wedge \psi_2$, we make exactly the same construction, only using an AND state as the new initial state.

If ϕ is $\exists x\psi$, and T is a tape assignment for the free variables of ϕ, then we construct $M(\psi)$ using the extended tape assignment which assigns a new tape $k + 1$ to the variable x, and otherwise is the same as T. Now M is constructed to go through an initial loop of existential states, which fills tape $k + 1$ with a string no longer than the maximum length of any string on tapes 1 through k. It then transfers control to the initial state of $M(\psi)$. The same construction is used for the universal quantifier, using an initial loop of universal states.

Finally, we need to treat the case of a recursion scheme $\mu S\Phi$. Suppose that Φ has domain $\mathcal{R}$, and let T be a tape assignment for $\mu S\Phi$. For each clause $C\Phi(Q)$, where $Q \in \mathcal{R}$, we construct a machine $M(Q)$ by inductive hypothesis. The global free variables of each $M(Q)$ will have tapes assigned by T. However, we construct the $M(Q)$ all in such a way that the local tape numbers do not overlap the tape numbers for any other $M(R)$. This procedure will give tape numbers to all the variables in the set $B\Phi(Q)$. Let this set be $\{z_1, \ldots, z_k\}$ in increasing order. Define $T_Q(z_i)$ to be the tape assigned to z_i in $M(Q)$.

The machine $M(\mu S\Phi)$ will consist of the code for the $M(Q)$, arranged as blocks; the initial state of each such block will be labeled Q. In all the blocks, recursive oracle calls to Q? will be replaced by statements transferring control to Q. Thus, consider an oracle call $Q?(i_1, \ldots, i_k)$, in any block $M(R)$. Replace this call by code which copies tape i_1 to tape $T_Q(z_1)$, . . ., and tape i_k to tape $T_Q(z_k)$. Insert code that empties all other tapes local to $M(Q)$, and insert a statement "go to Q."

This completes the construction, and we now illustrate it with an example. Consider the recursion scheme introduced in the first section.

$$S(x) \Leftrightarrow \exists y((x = ay \wedge (S(y)) \vee (x = by \wedge T(y))) \vee x = a$$

$$T(v) \Leftrightarrow \exists w(v = cw \wedge S(w)).$$

We construct the machine $M(S)$ as follows:[1]

tape 1 : x
tape 2 : y (initially blank)
Initially : guess a value of y, such that $|y| \leq |x|$, and store y on tape 2; go to (q1 or q2 or q7);
q1 : go to (q3 and q4);
q3 : check $x = ay$ on tapes 1 and 2, and accept or reject as appropriate;
q4 : S?(tape 2)
q2 : go to (q5 and q6);
q5 : check $x = by$ on tapes 1 and 2, and accept or reject as appropriate;
q6 : T?(tape 2)
q7 : check $x = a$ and accept or reject.

Similarly, we can construct a machine $M(T)$ for the T clause. Then the result of pasting together the two constructions is shown in Figure 1.

```
tape 1 : x
tape 2 : y (initially blank)
tape 3 : v (initially blank)
tape 4 : w (initially blank)

S : guess a value of y, such that |y| ≤ |x|, on tape
2;
go to (q1 or q2 or q7);
     q1 : go to (q3 and q4);
     q3 : check x = ay on tapes 1 and 2 , and
accept or reject as appropriate;
     q4 : copy tape 2 to tape 1;
          Empty tape 2;
          Go to S.

     q2 : go to (q5 and q6);
     q5 : check x = by on tapes 1 and 2 , and
accept or reject as appropriate;
     q6 : copy tape 2 to tape 3;
          empty tape 4;
          go to T.

     q7 : check x = a and accept or reject.

T : guess a w on tape 4 no longer than v on tape
3;
go to (q9 and q10);
     q9 : Check v = cw on tapes 3 and 4, and
     return appropriately;
     q10 : copy tape 4 (w) to tape 1;
           empty tape 2;
           go to S.
```

Figure 1. ATM Program for the Recursion Scheme.

As we remarked, we cannot give a full proof of the correctness of our construction. However, the construction does correspond to the formal semantics of CLFP. In particular, the semantics of recursion corresponds to the iterated schemes Φ^k. Iterating the scheme k times roughly corresponds to developing the computation tree of the ATM to k levels, and replacing the oracle states at the leaves of the k-level tree with rejecting states corresponds to substituting FALSE into the kth iteration.

With these remarks, the proof is complete.

Lemma 3. Suppose L is accepted by a $S(n) = n$-bounded ATM. Then there is a CLFP formula ϕ such that for all $u \in \Sigma^*$, we have $u \in L \Leftrightarrow u \models \phi$.

Proof: We may assume that M is an ATM with one work tape, if we allow M to print symbols in an auxiliary tape alphabet Γ. By a result in Chandra, Kozen, and Stockmeyer (1981) M has no negating states. We show how to construct a formula ϕ, which has constants ranging over Γ, but which has the property stated in the conclusion of the lemma: for each string x over Σ, M accepts x iff $x \models \phi$. The formula ϕ will be given as a recursion scheme $\mu S\Phi$. Each state q of M will become a binary predicate variable $q(x,y)$ in $\mathcal{R}$. The meaning of $q(u,v)$, where u and v are specific strings in Γ^*, is that M is in state q, scanning the first symbol of v, and that u and v are the portions of the work tape to the left and the right of the head, respectively.

We give a perfectly general example to illustrate the construction of Φ. In this example, the tape alphabet Γ is $\{a,b\}$. Suppose that q is a universal state of M and that $\delta(q,a) = \{(r,b,right),(s,a,left)\}$, and $\delta(q,b) = \{(p,b,left),(q,a,right)\}$. *Then* $\Phi(q)(x,y)$ is the following formula:

$$\bigwedge_{\sigma\in\{a,b\}} \forall wt[(x = w\sigma \wedge y = at \Rightarrow r(xb,t) \wedge s(w,\sigma at))$$
$$\wedge(x = w\sigma \wedge y = bt \Rightarrow p(w,\sigma bt) \wedge q(xa,t))]$$

The distinguished element of $\mathcal{R}$ is q_0, the start state of M. Notice that all predicate variables in R occur positively in Φ, and that the search for w and t is limited to strings no longer than the length of the original input to M. If q is an accepting state of M, then we have a clause in Φ of the form $q(x,y) \Leftrightarrow$ TRUE, where TRUE is some tautology.

Technically speaking, the explicit substitutions $r(xb,t)$ are not allowed in our formulas, but these can be expressed by suitable sentences like $(\exists z)(z = xb \wedge r(z,t))$, as remarked in the first section. The cases for $q(x,y)$ when x and y are null must also be handled separately because M fails if it tries to leave the original region.

Finally, we can obtain a formula over the constant alphabet Σ by a more complicated construction. If we encode Γ into Σ by a homomorphic mapping, then a machine N can be constructed to simulate M. N will have tape alphabet Σ, but will have a number n of work tapes bounded linearly by the constant involved in the encoding. We now make a formula corresponding to N, but the predicates will have to be $2n$-ary, one pair of arguments for each tape of N. With these remarks, the proof of the lemma is complete.

Theorem 1 follows immediately from the above lemmas.

4 ILFP: Grammars with Integer Indexing

4.1 Syntax of ILFP

Our characterization of the defining power of CLFP relied on the result $EXPTIME = ASPACE(n)$. We also know that $PTIME = ASPACE(\log n)$. Is there a similar logical notation that gives a grammatical characterization of PTIME? This section is devoted to giving an affirmative answer to this question. As stated in the introduction, this result is already known (Immerman 1982, Vardi 1982), but the result fits well with the CLFP theorem, and may in the linguistic domain have some real applications other than ours to Head Grammars. To explain the logic, it helps to consider acceptance by a logspace bounded ATM. In this case, the machine has a read-only input tape, which can be accessed by a two-way read head. Writing is strictly disallowed on the input tape, in contrast to the linear space bounded ATMs of the previous section. There is also a number k of work tapes on which computation occurs. Suppose that these work tapes use a binary alphabet. If their size always is less than or equal to $\lceil \log_2 n \rceil$, then they are always capable of representing the numbers from 0 through $n - 1$. We thus think of the contents of the work tapes as indices of specific positions in the read-only input string, though in fact they may not serve this purpose in an arbitrary computation. Since the input is off-line, substrings of the input will not be quantified. Instead, we quantify over the integer subscripts, and the input simply becomes a global parameter appearing in the semantics. Instead of having equations between strings as atomic formulas, we will have equations between integer terms. In order to access the input, we will have, for each symbol $a \in \Sigma$, an atomic predicate symbol $a(i)$ of one argument, which will be true iff in the given input x, the symbol $x(i)$ at position i is a. (We number the positions from 0 through $n - 1$). We allow individual constant symbols **0**,**1**, and **last**, which will be interpreted as 0, 1, and $n - 1$, respectively, when the input has size n. As primitive arithmetic operations we allow addition and subtraction, and multiplication and integer division by 2. All of these operations are interpreted modulo n when the input is given.

We need not give the formal definition of ILFP formulas, as it is the same as for CLFP, except that individual variables come from a set $\{i_0,i_1, \ldots\}$, terms are formed as above from arithmetic combinations of individual variables and constants, and the unary predicates $a(i)$ are atomic formulas.

Example 2. Consider the CFG

S → aSb | bSa | SS | ab | ba

This is represented in ILFP as follows:

$$S(i,j) \Leftrightarrow a(i) \wedge S(i+1,j-1) \wedge b(j)$$
$$\vee\, b(i) \wedge S(i+1,j-1) \wedge a(j)$$
$$\vee\, \exists k < j : S(i,k) \wedge S(k+1,j)$$
$$\vee\, j = i+1 \wedge ((a(i) \wedge b(j)) \vee (b(i) \wedge a(j))$$

(Again, the explicit substitution of terms for variables is not officially allowed but can be introduced by definition.)

The meaning of the above scheme should be clear. The predicate $S(i,j)$ is intended to mean that node S dominates positions i through j in the input. Thus the assertion $S(0,\textbf{last})$, with no free variables, will be satisfied by a string x iff x is generated by the given CFG. The relation of this descriptive formalism to the CKY algorithm for context-free recognition should also suggest itself.

Our definition of the meaning function $\mathcal{M}[\![\phi]\!]$ is like that in Section 2, except that the parameter n is replaced by a string $x \in \Sigma^*$. Thus

1. $\mathcal{M}[\![p(i_1, \ldots, i_k)]\!]\rho x = \{\alpha \mid \langle\alpha(i_1), \ldots \alpha(i_k)\rangle \in \rho(p)\}$;
2. $\mathcal{M}[\![a(i)]\!]\rho x = \{\alpha \mid x(\alpha(i)) = a\}$;
3. $\mathcal{M}[\![t_1 = t_2]\!]\rho x = \{\alpha \mid t_1\alpha = t_2\alpha\}$;
4. $\mathcal{M}[\![\exists i\phi]\!]\rho x = \{\alpha \mid (\exists m < |x|)(\alpha(i/m) \in \mathcal{M}[\![\phi]\!]\rho x)\}$;
5. Boolean combinations are as before;
6. $\mathcal{M}[\![\mu S\Phi]\!]\rho x = \{\alpha \mid (\exists k)(\alpha \in \mathcal{M}[\![C\Phi^k(S)]\!]\rho x)\}$

The schemes Φ^k are defined for recursion schemes as above.

If ϕ is a formula of ILFP with no free individual or predicate variables then $S[\![\phi]\!]\rho x$ is either A, the set of all individual assignments, or $\emptyset$, independent of ρ, but depending on x. We say that $x \models \phi$ iff $S[\![\phi]\!]\rho x$ is all of A. A language $L \subseteq \Sigma^*$ is ILFP-definable iff for some ϕ in ILFP, $L = \{x \mid x \models \phi\}$. Our objective is now

Theorem 2. A language is ILFP-definable iff it is in *PTIME*.

The proof appears in the next subsection.

4.2 PROOF OF THEOREM 2

The idea of our proof is the same as that for Theorem 1, and only a sketch of the proof is necessary. We first restate Lemma 2 for ILFP, using the same definition for β and γ.

Lemma 4. Let ϕ be an ILFP formula, with $|FIvar(\phi)| = k$, and $T : FIvar(\phi) \rightarrow \{1, \ldots, k\}$. Let $m = \gamma(\phi)$. Then we may construct an m-tape ATM $M(\phi,T)$ having the following properties: (i) M has oracle states P? for each free predicate variable of ϕ, and (ii) For any $x \in \Sigma^*$, any α mapping $FIvar(\phi)$ to natural numbers, and any environment ρ, we have the following: M with oracle states for the $\rho(P)$, started with x on the input tape, binary representations of the integers $\alpha(i_1)$ on tape $T(i_1)$, . . ., and $\alpha(i_k)$ on tape $T(i_k)$, and the other tapes blank, will accept without ever writing a value $j > |x|$ on any tape, if and only if $\langle\alpha(i_1), \ldots, \alpha(i_k)\rangle \in \mathcal{M}[\![\phi]\!]\rho x$.

Proof: The proof is almost identical to that of Lemma 2. To evaluate equations M may have to use an extra tape, because otherwise the given nonblank tapes would be overwritten by the arithmetic operations. If ϕ is $a(i)$ (the only case not covered in (2), then tape 1 is used as a counter to locate the input head at the position of the contents of tape 1. Since arithmetic is modulo $|x|$, the machine never writes too great a value in these cases. The other cases are proved exactly as in (2), so this completes the proof.

Lemma 5. If $L \in ASPACE(\log n)$, then L is ILFP-definable.

Proof: We may assume that L is accepted by an ATM with p binary work tapes and one input tape. (If the tape alphabet is not binary, encode with a homomorphism and expand the number of tapes as necessary.) We may further assume that the machine M never writes a string longer than $\lfloor \log_2(n) \rfloor - 1$ on any work tape (remember one bit on each tape in finite control if necessary). Each work tape, or portion thereof, is thus guaranteed to represent a binary number strictly less than n in value, where n is the length of the input string.

We now proceed as in the proof of Lemma 3, but coding the contents of the work tapes as binary numbers. We need a number h, which tells the position of the input head. We also have two numbers l and r, which are the binary values of the tape contents to the left and right of the work tape head (here we describe the case of just one work tape). The number r will actually be the binary value of the reversal of the string to the right of the tape head, because this makes the operation of shifting the head a simple multiplication or division by 2. Since a string may have leading zeroes, we also need to keep two auxiliary numbers *ll* and *rr*, which are the actual lengths of the strings to the left and right of the head. For each state q of the ATM we thus have a predicate $q(h,l,r,ll,rr)$ of five integer variables. The reader should have no difficulty in encoding the transition rules of M exactly as in Lemma 3. For example, a test as to whether the scanned symbol on the work tape is 0 or 1 becomes a test of the parity of r, and so on. Finally, it can be seen that the case of p work tapes requires $4p + 1$-ary predicates. This completes the proof of our lemma and thus the theorem.

4.3 WHICH POLYNOMIAL?

We can get a rough estimate of the degree of the polynomial time algorithm, which will recognize strings in the language defined by an ILFP grammar. We saw in the proof of Lemma 4 that if a scheme ϕ has $\gamma(\phi) = p$, then an ATM with $p + 1$ binary work tapes can be constructed to recognize the associated language. The number of configurations of each tape is thus $\log n * 2^{\log n+1}$. If there are $p + 2$ tapes, this gives $O(\log^{p+1} n * n^{p+1}) = O(n^{p+2})$ possible tape configurations. Multiplying by n for the position of the input head gives $O(n^{p+3})$ possible ATM configurations. From an analysis of the proof of Lemma 1 in Chandra, Kozen, and Stockmeyer

(1981), we can see that the polynomial in our deterministic TM algorithm is bounded by the square of the number of ATM configurations. This leads to an $O(n^{2p+6})$ recognition algorithm. Since this bound would give an $O(n^{12})$ algorithm for context-free language recognition, we conjecture that the general estimate can be improved. In particular, we would like to remove the factor of 2 from $2p$.

5 Applications to Head Grammars

In this section we express head grammars (Pollard 1984) in ILFP, and thus show that head languages can be recognized in polynomial time. Since the class of head languages is the same as the class of tree adjunct languages (Vijayashankar, Joshi 1985), we get the same result for this class. We will actually give only a simplified version of head grammars to make our ILFP formulas easy to write. This version corresponds exactly to the Modified Head Grammars of Vijayashankar and Joshi (1985), and differs only from the original version in that it does not treat the empty string. (Roach (1988) has an extended discussion of head languages.)

We define a head grammar as a tuple $G = \langle N,\Sigma,P,S\rangle$, where N and Σ are finite nonterminal and terminal alphabets, P is a finite set of productions, and S is the start nonterminal. The productions are of the form $C \rightarrow \mathrm{Op}(A,B)$, where A,B, and C are nonterminals and Op is chosen from a fixed set of **head-wrapping operations**. Productions can also have the form $C \rightarrow (x,y)$, where x and y are terminal strings.

We view nonterminals in N as deriving pairs of strings (u,v). In the original formulation, this meant that the head of the string uv occurred at the end of u. The wrapping operations come from the set $\{LL_1,LL_2,LC_1,LC_2\}$. We consider LL_2 and LC_1 as examples. We define $LL_2((w,x),(u,v)) = (wu,vx)$. Thus if A derives (w,x) and B derives (u,v), and $C \rightarrow LL_2(A,B)$ is a production, then C derives (wu,vx). Similarly, $LC_1((w,x),(u,v)) = (w,xuv)$, so in the corresponding case, we would have C derives (w,xuv) if $C \rightarrow LC_1(A,B)$ were a production. A string t is in $L(G)$ iff for some u and v, $t = uv$ and S derives (u,v).

Given a head grammar, we write an ILFP recursion scheme as follows. For each nonterminal C, we introduce a predicate $C(i,j,k,l)$. We think of these four integers as indexing the positions of symbols in a string, starting at the left with 0. Then $C(i,j,k,l)$ means that the nonterminal symbol C can derive the pair of substrings of the input string between i and j, and between k and l inclusive. Thus, if $C \rightarrow LL_2(A,B)$ is a production, our scheme would include a clause

$$C(i,j,k,l) \Leftrightarrow (\exists pq)(A(i,p,q+1,l) \wedge B(p+1,j,k,q))$$

Similarly, if $C \rightarrow LC_1(A,B)$ were a production, we would have

$$C(i,j,k,l) \Leftrightarrow (\exists pq)(A(i,j,k,p) \wedge B(p+1,q,q+1,l))$$

Finally, if $C \rightarrow (a,bb)$ were a terminating production, we would have

$$C(i,j,k,l) \Leftrightarrow a(i) \wedge i = j \wedge k = i+1 \wedge b(k) \wedge b(k+1) \wedge l = k+1$$

The grammar would be defined by the recursion scheme and the assertion $\exists jS(\mathbf{0},j,j+1,\mathbf{last})$, where S is the start symbol of G.

It can be seen from this formulation that every head grammar can be written as an ILFP scheme with at most six total variables. Section 4 thus gives us an $O(n^{18})$ algorithm. However, the algorithm of Vijayashanker and Joshi (1985) is at most n^6. It would seem that a rule of thumb for the order of the polynomial algorithm is to use the number $\gamma(\phi)$ for the ILFP scheme ϕ, but we have no proof for this conjecture.

Acknowledgment

Research supported by NSF Grant MCS-8301022.

References

Chandra, A.K. and Harel, D. 1982 Structure and Complexity of Relational Queries. *Journal of Computer Systems Science* 25: 99–128.

Chandra, A.K.; Kozen, D.C.; and Stockmeyer, L.J. 1981 Alternation. *Journal of Associated Computer Machines* 28: 114–133.

Immerman, N. 1982 Relational Queries Computable in Polynomial Time. In *Proceedings of the 14th ACM Symposium on Theory of Computing*: 147–152.

Johnson, M. 1985 Parsing with Discontinuous Constituents. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, Chicago, IL: 127–132.

Pereira, F.C.N. and Warren, D.H.D. 1980 Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13: 231–278.

Pollard, C. 1984 Generalized Phrase Structure Grammars, Head Grammars, and Natural Language. Ph.D thesis, Stanford University, Stanford, CA.

Roach, K. (1988) Formal Properties of Head Grammars. Manuscript, Stanford University. In Manaster-Ramer, A. (ed.) *Mathematics of Language*. John Benjamins, Amsterdam, Holland.

Shapiro, E.Y. 1984 On the Complexity of Logic Programs. *Journal of Logic Programming* 1.

Vardi, M. 1982 The Complexity of Relational Query Languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing*: 137–146.

Vijayashankar, K. and Joshi, A.K. 1985 Some Computational Properties of Tree Adjoining Languages. In *Proceedings of the 23rd Meeting of the Association for Computational Linguistics*, Chicago, IL: 82–93.

Note

1. Notice that the machines are presented in ATMGOL, a syntactically ill-defined variant of ATM transition functions. Also, ATMs and ATNs are not to be confused.